SCHAUN

C für Einsteiger

while
if
struct
case
continue

Ein **DATA BECKER** Buch

AMIGA

SCHAUN

C für Einsteiger

while if struct case continue

Ein **DATA BECKER** Buch

AMIGA

ISBN 3-89011-107-6

2. erweiterte Auflage

Copyright © 1987 DATA BECKER GmbH
Merowingerstraße 30
4000 Düsseldorf

Alle Rechte vorbehalten. Kein Teil dieses Buches darf in irgendeiner Form (Druck, Fotokopie oder einem anderen Verfahren) ohne schriftliche Genehmigung der DATA BECKER GmbH reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.*

Wichtiger Hinweis:
Die in diesem Buch wiedergegebenen Schaltungen, Verfahren und Programme werden ohne Rücksicht auf die Patentlage mitgeteilt. Sie sind ausschließlich für Amateur- und Lehrzwecke bestimmt und dürfen nicht gewerblich genutzt werden.

Alle Schaltungen, technischen Angaben und Programme in diesem Buch wurden von dem Autoren mit größter Sorgfalt erarbeitet bzw. zusammengestellt und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. DATA BECKER sieht sich deshalb gezwungen, darauf hinzuweisen, daß weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgen, die auf fehlerhafte Angaben zurückgehen, übernommen werden kann. Für die Mitteilung eventueller Fehler ist der Autor jederzeit dankbar.

Inhalt

1. Grundsätzliches zu C 13
1.1 Prinzip der Programmausführung 14
1.2 Compiler oder Interpreter 15

Teil 1 - C an einem Wochenende 17

2. Der Einstieg in C 18
2.1 Der Editor 18
2.2 Der C-Compiler 19
2.3 Der Linker 20
2.4 How to use 21

3. Das erste Programm 22
3.1 ED im Einsatz 23
3.2 Compiler im Einsatz 24
3.3 Klappt es nicht? 26

4. Theorie und Praxis 28
4.1 Die Formatierung des Programms 29
4.2 Definition einer Funktion 31
4.3 printf und Steuerzeichen 32
4.4 Kommentare 33
4.5 Erste Bekanntschaft mit Variablen und Arithmetik in C 35
4.5.1 Integer 35
4.5.2 Wenn, dann... - der if-Befehl 37
4.5.3 Rechnen mit C 41
4.5.4 Fließkommazahlen 44
4.5.5 Zeichen und Zeichenketten 45

5. Schleifen und was dazu gehört 51
5.1 while-Schleifen 51
5.2 Die for-Schleife 53
5.3 Die do-while-Schleife 54
5.3.1 Schnitzeljagd 55
5.4 UND oder ODER 58

6. Strings durchleuchtet 61
6.1 Rückwärts geht's weiter 62

Teil 2 - C durch und durch 65

7. Rechnen in C 66

8. Variablen 68
8.1 Variablennamen 68
8.2 Datentypen 70
8.3 Typumwandlungen 74
8.4 Die cast-Anweisung 74

9. printf und scanf 76
9.1 Steuerzeichen 76
9.2 Formatanweisungen bei printf und scanf 77
9.3 Oktal und Hexadezimal 82
9.3.1 Kleines Umrechenprogramm 85
9.4 Code eines Zeichens 87
9.4.1 Der Backslash hilft 90
9.4.2 In die andere Richtung 91

10. Der Präprozessor 92
10.1 #define 92
10.2 #include 95

11. Abkürzungen 97
11.1 Inkrement und Dekrement 99
11.2 Initialisierung, Definition, Deklaration 101
11.3 Mehrfachzuweisungen und Wert eines Ausdrucks in C 103

12. Funktionen 105
12.1 Funktionen mit Parametern 106
12.2 Funktionen ohne Rückgabewert 107
12.3 Eigene Funktionen 108
12.3.1 strcpy-Version 1 109
12.3.2 strlen 111

13. Arrays ... 113
13.1 Mehrdimensionale Felder ... 114

14. Schleifen ... 120
14.1 Feinheiten der for-Schleife ... 120
14.2 break ... 121
14.3 continue ... 122
14.4 Die switch-Anweisung ... 123

15. Zeiger und Adressen ... 125
15.1 Adresse ... 125
15.2 Zeiger oder Pointer ... 126
15.2.1 Tauschfunktion mit Zeigern ... 128
15.2.2 strcpy Version 2 ... 130
15.2.3 strcpy Version 3 ... 130
15.3 Zeiger ohne Speicher ... 133

16. Speicherklassen ... 135
16.1 auto ... 135
16.2 static ... 135
16.3 global ... 136
16.4 register ... 137
16.4.1 Schnelle strcpy-Routine ... 138
16.5 Lokal ... 141

17. Unsere Bibliothek ... 143
17.1 strcmp ... 145
17.2 itoa ... 149
17.3 reverse ... 151

18. C-Features ... 153
18.1 ?:-Operator ... 153
18.2 sizeof ... 154
18.3 Bitmanipulationen ... 155
18.3.1 UND ... 155
18.3.2 ODER ... 156
18.3.3 Bitgeschiebe ... 159
18.3.4 EXKLUSIVE-ODER ... 160
18.3.5 Komplement-Operator ... 160
18.4 goto ... 161

19. Zusammengesetzte Datentypen ... 163
19.1 Eine Struktur ... 163
19.2 Bitfelder ... 166
19.3 union ... 167
19.4 Aufzählung: enum ... 168
19.5 typedef ... 169

20. Wichtige Grundbegriffe ... 170
20.1 Deklarationen ... 170
20.2 Initialisierungen ... 173

21. Zeiger-Arrays ... 176

22. Nützliche Makros ... 180
22.1 Fehlerquellen bei Makros ... 181
22.2 Makros für die Bibliothek ... 184

23. Kontakt mit der Außenwelt ... 187
23.1 Datenübergabe mittels Kommandozeile ... 188
23.2 Ein-/Ausgabe ... 190
23.2.1 Gepufferte Ein-/Ausgabe ... 191
23.3 Weitere gepufferte Ein-/Ausgabefunktionen ... 198
23.4 Ungepufferte Ein-/Ausgabefunktionen ... 202
23.5 Direktzugriff ... 205
23.6 Einlesen eines Zeichens ... 207
23.6.1 Standardein- und -ausgabe ... 208
23.7 Ein eigenes Fenster ... 209
23.7.1 Die drei Modi ... 210
23.8 Umleitungen ... 213

24. Tips und Tricks ... 217
24.1 Starten von der Workbench ... 217
24.2 Weitere Anweisungen des Präprozessors ... 220
24.3 Fehlerursachen und deren Beseitigung ... 222

25. Systemprogrammierung 225
25.1 Das Intuition-Prinzip 225
25.2 Ein Window unter Intuition 226
25.2.1 Die Window-Flags 228
25.2.2 Öffnen eines Windows 230
25.2.3 Ein Window-Programm 231
25.3 Screens 234
25.3.1 Ein Screen-Programm 237
25.4 Text- und Grafikausgabe in ein Fenster 239
25.4.1 Text 240
25.4.2 Move 240
25.4.3 Draw 241
25.4.4 Kleines Malprogramm 242
25.4.5 Niedrige Auflösung und Interlace 247
25.4.6 Pixelbearbeitung 249
25.5 DOS 253
25.6 SetComment 254
25.7 Inhaltsverzeichnis auslesen 255

Anhang

A: Funktionen 260
B: Die Geschichte von C 266
C: Der Lattice-C-Compiler 269
D: Aztek-C 275
E: Reservierte C-Befehlswörter 281
F: Prioritäten und Reihenfolge der C-Operatoren 282
G: Speicherklassen 283
H: Typumwandlungen 284
I: Modi für fopen (beim Lattice-C) 285
J: Stichwortverzeichnis 287

1. Grundsätzliches zu C

Sie wollen C lernen, sonst würden Sie ja auch nicht dieses Buch in den Händen halten. Un genau das ist auch das Ziel dieses Buches. In möglichst kurzer Zeit sollen Sie lernen, in C zu programmieren. Um dieses Ziel schnell zu erreichen, teilt sich das Buch grob in zwei Teile. Im ersten Abschnitt lernen Sie anhand einfacher Beispielprogramme die Grundstrukturen eines C-Programms kennen. So sind Sie innerhalb kürzester Zeit in der Lage, selbst eigene kleinere Programme in C zu schreiben. Im zweiten Teil geht es dann ans Eingemachte, wo dann auch auf alle Besonderheiten und Hintergründe der einzelnen Befehle bei der C-Programmierung eingegangen wird.

Der schnelle Einstieg steht im Vordergrund des ersten Abschnitts. Das obligatorische Kapitel über Geschichte und Entwicklung der Sprache C findet der interessierte Leser im Anhang. Das gleiche gilt auch für die Bedienungsanleitung der einzelnen C-Compiler. Wer sich bereits im Umgang mit Editor, Compiler und Linker auskennt, braucht also keine Seiten zu überschlagen. Derjenige, der sich aber mit diesen Programmen noch vertraut machen möchte, schlägt dem Compilertyp entsprechend im Anhang nach. Die Theorie beschränkt sich im Einstiegsteil also auf das absolut Notwendigste.

Aber bevor es so richtig losgeht, wollen wir uns doch einmal die Frage stellen, was C eigentlich ist.
Eine Antwort darauf wäre: C ist eine Compilersprache. Man unterscheidet nämlich zwei Gruppen von Computersprachen. Zum einen sind da eben die Compilersprachen wie C, Pascal oder Modula2, zum anderen gibt es Interpreter z.B. für BASIC oder LOGO. Was sind nun aber Compiler und Interpreter und welche Vor- und Nachteile bringen sie uns?

1.1 Prinzip der Programmausführung

Compiler sind Programme, die die eingegebenen Befehle der benutzten Sprache in eine für den Computer verständliche Form übersetzen. Diese Form besteht nur aus den Ziffern '0' und '1', den sogenannten Binärzahlen, die einzigen Dinge, mit denen ein Computer wirklich etwas anfangen kann. Da sich der Mensch keine langen Kolonnen von Nullen und Einsen merken kann, greift er zu einer Methode, die es für ihn vereinfacht, dem Rechner verständlich zu machen, was er tun soll. Er definiert für einzelne vom Rechner ausführbare Binärkombinationen kurze Wörter oder Abkürzungen, die genau das beschreiben, was die Maschine bei Erhalt einer solchen Binärzahl machen soll. So bedeutet z.B. das Kürzel "LDA" "Load Akkumulator", das natürlich besser zu behalten ist, als eine Binärzahl wie 10011101.

Bei diesem Verfahren muß dem Rechner aber alles so präsentiert werden, daß er zu jedem Kurzbefehl einen Maschinenbefehl parat hat. Für den Menschen bedeutet dies aber eine Menge Arbeit, da praktisch jeder Schritt einzeln programmiert werden muß. Diese Programmierung mit den Binärziffern nennt man Maschinensprache, oder in der Schreibweise mit den Abkürzungen: Assemblersprache. Allerdings unterscheiden sich die Befehle von Rechner zu Rechner. Die Instruktion, die ein Prozessor (das Gehirn des Rechners) ausführen kann, existiert vielleicht auf dem anderen überhaupt nicht.

Es wäre natürlich für den Programmierer viel komfortabler, wenn man dem Rechner in der Umgangssprache sagen könnte, welche Aufgaben zu lösen sind. Soweit ist man aber bei der Entwicklung von Computersprachen noch nicht. Einen Kompromiß zwischen der Maschinen- und der Umgangssprache bieten die sogenannten Hochsprachen. In diesen Sprachen existiert eine begrenzte Zahl von Wörtern, die eine bestimmte festgelegte Funktion ausführen, die aber im Gegensatz zur Assemblersprache nicht unbedingt auf die Maschine zugeschnitten sein muß. Der Rechner muß dann bisweilen mehrere hundert Maschinenbefehle für einen einzigen Befehl einer Hochsprache ausführen. Ein Beispiel wäre der BASIC-Befehl "LOAD Dateiname", der den Computer zum Laden einer Datei veranlaßt.

1.2 Compiler oder Interpreter

Hier scheiden sich nun die Wege des Compilers und des Interpreters. Der Interpreter sucht aus dem Programmtext die Befehlsworte heraus, prüft, ob es ein erlaubter Befehl ist, und führt anschließend die entsprechenden Maschinenbefehle aus. Dann holt er sich den nächsten Befehl, prüft und führt wieder die Maschinenbefehle aus. Dies macht er immer wieder, auch, wenn er den Teil schon in Maschinensprache übersetzt hat. Da wir schon beim Übersetzen sind, sei ein Vergleich zum Dolmetscher gestattet:

Ein Interpreter ist praktisch ein Simultandolmetscher, der die Befehlsworte der Hochsprache in Maschinenbefehle übersetzt. Im Gegensatz zum Interpreter übersetzt der Compiler jedes Programm ein einziges Mal, vergleichbar mit einem Übersetzer für fremdsprachige Literatur. Dieser kann sich beim Übersetzen viel Zeit lassen, die Wahl der Worte viel genauer abwägen als der Simultandolmetscher und bereits übersetzte Zusammenhänge übertragen. Der daraus resultierende Vorteil des Interpreters zum Compiler ist, daß man bei einem Interpreter sofort eine Reaktion vom Rechner erhält, wenn Sie etwas eingeben. Sie können den Interpreter auch einfach unterbrechen, beispielsweise nachsehen, welche Werte in bestimmten Variablen abgespeichert sind, und abschließend den Programmablauf fortsetzen, ohne daß dadurch der Interpreter in Schwierigkeiten kommt. Der Vorteil des Interpreters ist also seine Flexibilität und Spontanität.

Die Stärken des Compilers liegen aber ganz woanders. Dadurch, daß das gesamte Programm einmal übersetzt werden muß, braucht der Compiler vor dem Starten des Programms vielleicht ein paar Minuten. Danach kann das Programm aber viel schneller abgearbeitet werden, da kein Interpreter ständig neu übersetzen muß und den Ablauf des Programmes verzögert. Einleuchtend wird dieser Umstand besonders bei Schleifendurchläufen, bei dem ein Befehl mehrere tausendmal ausgeführt werden soll. Der Interpreter schaut nach, was er tun soll, übersetzt diesen Befehl in Maschinensprache und macht das Ganze

noch mehrere tausendmal. Das bereits von einem Compiler übersetzte Programm weiß, was es zu tun hat, ohne daß es ihm jedesmal neu erklärt werden muß. Der Vorteil eines Compilers liegt also in der wesentlich schnelleren Ausführungsgeschwindigkeit eines Programmes.

Hinzu kommt auch noch ein psychologischer Aspekt, nämlich, daß der Programmierer in einer Interpretersprache dazu verleitet werden kann, einfach drauflos zu programmieren. Es wird einfach ausprobiert, bis zu welcher Stelle ein gerade am Bildschirm entwickeltes und frisch eingetipptes Programm fehlerfrei läuft, anstatt vorher zu prüfen, wo Fehler auftreten könnten. Beim Compiler ist das nicht so einfach, da nach jeder Fehlermeldung des Compilers erst wieder das Programm korrigiert werden und der Compiler sich von neuem ans Übersetzen machen muß, was ja mehrere Minuten in Anspruch nehmen kann. Eine dieser Compilersprachen ist C, und zwar eine, die ganz besonders fix in der Ausführung eines Programmes ist, wie Sie noch sehen werden.

C an einem Wochenende

2. Der Einstieg in C

Bevor wir nun aber beginnen können, sollten Sie sich den Werdegang eines C-Programmes zu Gemüte führen. Den folgenden Brocken Theorie kann ich Ihnen leider nicht ersparen. Sie bildet die Grundlage beim Programmieren in C, ohne deren Verständnis Sie wohl kein Programm zum Laufen bringen. Das Prinzip, wie man von der Idee zum fertigen C-Programm gelangt, ist folgendes:

Zuerst muß der Programmablauf aus der Idee entwickelt werden, wie es bei anderen Sprachen auch der Fall ist. Nachdem Sie wissen, wie Ihr Programm aussehen soll, können Sie sich an den Rechner setzen.

2.1 Der Editor

Um ein C-Programm einzutippen, benötigen Sie einen Texteditor. Ein Editor ist nichts anderes als eine primitive Textverarbeitung. Er enthält meistens nur ganz geringe Möglichkeiten, den Text zu bearbeiten. Es wird aber von diesem Programm auch nicht mehr verlangt, als daß man den Programmtext eingeben, laden, speichern und verändern kann. Für die Aufgabe, ein C-Programm einzutippen, können Sie natürlich auch eine ganz normale Textverarbeitung heranziehen. Es ist lediglich darauf zu achten, daß im Programmtext keine speziellen Steuerzeichen z.B. für Fettschrift oder Textformatierung vorkommen, die der Übersetzer (der Compiler) nicht versteht. Viele Textverarbeitungen bieten deshalb einen speziell dafür vorgesehenen Modus an, eine Datei als "ASCII-Datei" abzuspeichern.

Außerdem muß die Textverarbeitung die Möglichkeit vorsehen, amerikanische Sonderzeichen einzugeben. Für ein C-Programm benötigen Sie nämlich die geschweiften '{}' und eckigen Klammern '[]', den sogenannten Backslash '\', das Doppelkreuz '#' und Sonderzeichen wie '|' und '~'. Vielleicht benutzen Sie ja den Editor "ED", der auf der Workbench-Diskette des AMIGA enthalten und gut geeignet ist.

Der Editor wird geladen und das C-Programm eingetippt. Diesen Text, auch C-Source (source (engl.) = Quelle) genannt, speichert man unter einem aussagekräftigen Namen mit der Endung ".C" auf Diskette ab. Aussagekräftig soll heißen, daß Sie Ihre Programmdateien nicht "A.C" oder "DINGSDA.C", sondern z.B. "SORT.C" oder "ARCHIV.C" nennen, also Namen wählen, die Rückschlüsse auf ihre Funktion zulassen. Besonders wichtig ist auch, daß auf jeden Fall die Kennung ".C" angehängt wird, da mehrere Dateien mit dem gleichen Namen, aber unterschiedlicher Endung erzeugt werden. Wenn eine solche Datei erstellt ist, kann der C-Compiler aufgerufen werden.

2.2 Der C-Compiler

Der vergebene Dateiname wird dem Compiler übergeben, und er beginnt mit der Übersetzung. Sollte der Compiler auf Befehle oder Ausdrücke stoßen, die er nicht versteht, so meldet er sich zu Wort. Fehlermeldungen werden auf dem Bildschirm und/oder in einer speziellen Fehlerdatei ausgegeben. Das hängt von dem verwendeten Compiler ab. Sind dem Compiler Fehler aufgefallen, was beim ersten Durchlauf eigentlich immer eintritt, dann muß der Editor erneut geladen und die vom Compiler gewünschten Korrekturen müssen durchgeführt werden. Sind alle Fehler beseitigt, wird die Datei erneut abgespeichert und der Compiler mit der Übersetzung beauftragt. Sollte er wieder etwas an Ihrem Programm auszusetzen haben, müssen Sie die ganze eben geschilderte Prozedur erneut durchführen.

Dieser Umstand erfordert deshalb vor dem Eintippen des C-Programms eine gute Vorbereitung, da Sie sich sonst eher mit dem Editor und den Fehlermeldungen des Compilers vertraut machen als mit der Sprache C.

2.3 Der Linker

Sollte der Compiler nach diesen Strapazen mit Ihrem Ergebnis zufrieden sein, so finden Sie auf der Compiler-Diskette eine Datei vor, die die Endung ".O" (Objectcode) trägt. Diese Datei kann noch nicht gestartet werden, sondern muß erst von einem weiteren Programm, dem .i.Linker bearbeitet werden. Der Linker (engl. Binder) sucht alle benötigten Funktionen für das Programm aus den Bibliotheken und bindet (deshalb auch der Name) diese zu einem Programm zusammen.

Eine Funktion ist ein Unterprogramm, und wird in Pascal z.B. als "PROCEDURE" bezeichnet. Die Unterprogramme sind in der Lage kleinere Aufgaben zu lösen, z.B. eine Linie zu ziehen oder einen Buchstaben auf dem Bildschirm auszugeben. In den Bibliotheken, die im Lieferumfang der C-Compiler enthalten sind, befinden sich diverse Funktionen, die oft gebraucht werden und bereits in übersetzter Form zur Verfügung stehen. Zu diesen Funktionen gehören die Ein-/Ausgabefunktionen, Grafikroutinen oder auch trigonometrische Funktionen. Aus den Bibliotheken entnimmt der Linker dann die Funktionen oder Routinen, die er für ein komplettes Programm benötigt. Dadurch wird dem Programmierer eine Menge Arbeit abgenommen. Er muß diese Funktionen nicht mehr selbst schreiben, sondern braucht sie lediglich mit den gewünschten Übergabewerten zu versorgen. Schließlich muß das Rad nicht zweimal erfunden werden.

Der Linker ermöglicht es, größere Programme modular, d.h. in mehreren Teilen getrennt, zu entwickeln. Jedes Modul wird separat compiliert und ausgetestet. Das hat den Vorteil, daß nicht jedesmal das gesamte C-Programm geladen und diverse Male übersetzt werden muß, wenn der Compiler einige Fehler findet. Danach können dann vom Linker alle übersetzten Module zu einem einzigen kompletten Programm zusammengebunden werden.

2.4 How to use

Alle oben aufgeführten Eingaben, um den Editor, den Compiler oder den Linker aufzurufen, werden mit Hilfe des CLI eingeben. So könnte beispielsweise der Linker durch folgende Zeile aufgerufen werden:

```
ALINK datei1.o datei2.o TO komplett
```

Obwohl wir zwar noch kein einziges Programm geschrieben haben, wissen Sie ja noch, daß ein Programm, bevor es fehlerfrei (syntaktisch, nicht logisch) ist, einige Male compiliert werden muß. Sie werden bestimmt einsehen, daß das direkte Eingeben einen enormen Arbeitsaufwand darstellt und ziemlich fehleranfällig ist. Das haben auch die Entwickler des C-Compilers berücksichtigt.

Es gibt deshalb die Möglichkeit, alle Eingaben in eine Datei zu schreiben. Aus dieser sogenannten MAKE-Datei werden dann nach und nach alle Programme wie Compiler oder Linker aufgerufen, als würden Sie alles selbst über die Tastatur eingeben. Eine MAKE-Datei zu erstellen, ist denkbar einfach. Anstatt alle Compiler-Aufrufe direkt ausführen zu lassen, schreiben Sie diese mittels Editor in eine eigene Datei und speichern sie auf Diskette ab.

Ein spezielles Programm (EXECUTE) des AMIGA-DOS liest anschließend diese Datei aus und leitet diese Informationen an die entsprechenden Compiler-Teile weiter. Im Anhang finden Sie eine solche MAKE-Datei, die Ihnen alle Aufrufsequenzen abnimmt.

Nachdem Sie nun doch einiges an Theorie verabreicht bekommen haben, soll endlich in "medias res" gegangen werden. Am besten lernt man eine Sprache, indem man sie spricht oder benutzt, deshalb präsentiere ich Ihnen zuerst einmal das erste Programm, welches dann ausführlich erläutert wird. Das folgende C-Programm gibt auf dem Bildschirm "lediglich" einen Text aus.

3. Das erste Programm

```
main()
{
  printf("Hallo, hier bin ich!");
}
```

Damit Sie sehen können, was das Programm produziert, müssen Sie obigen Programmtext mit dem Editor eintippen. Bitte übernehmen Sie anfangs den Text genau so, wie er hier abgedruckt ist. Dadurch werden Fehlermeldungen vermieden, die Sie vielleicht vor Probleme stellen könnten. Später, wenn Sie sich schon "fitter" in C fühlen, empfehle ich Ihnen sogar, Programme, nachdem (!) sie einwandfrei funktionieren, nach eigenem Ermessen zu verändern. Nur so kann man sich schnell die eine oder andere Frage selbst beantworten.

Jetzt aber Schritt für Schritt. Sollten Sie sich auf der Workbench befinden, so müssen Sie zuerst das Programm CLI aktivieren, das sich auf der Workbench-Diskette in der Schublade "SYSTEM" befindet. Sollten Sie es dort nicht entdecken können, dann ist das CLI bestimmt durch die Einstellung in Preferences abgeschaltet. Dann müßten Sie zuerst mit diesem Programm den Schalter "CLI" von OFF nach ON schieben. Und schon taucht es dann auch (hoffentlich) im Fenster auf. Wenn Sie das CLI gestartet haben, erhalten Sie ein neues Fanster, und zwar das CLI-Window. Es fordert Sie mit der Anzeige

```
1>
```

zur Eingabe auf.

Für einen Supercomputer wie den AMIGA ist dieses Programm vielleicht etwas ungewöhnlich. Keine Symbole (Icons), keine Mausfunktionen, es ist nur noch eine Bedienung über die Tastatur. Die Maus können Sie nun zum Abschied noch einmal benutzen, bevor sie irgendwo hinter dem Rechner verschwinden kann. Vergrößern Sie das CLI-Window mit Hilfe des Größensymbols auf das Maximum. Alles, was sich jetzt bei uns noch abspielt, landet hier im CLI-Fenster. Nun aber Maus zur Seite!

3.1 ED im Einsatz

Jetzt wird der Editor ED, der sich im Unterverzeichnis /C befindet, mit folgender Sequenz aufgerufen:

```
ED HALLO.C
```

In dieser Zeile steht vor dem "ED" natürlich noch das "1>", welches der Rechner dort plaziert hat. Diese Meldung, auch Prompt genannt, wird im weiteren nicht mehr erwähnt, sondern wir drucken nur noch Ihre Eingaben ab. Der Name "HALLO.C" ist der Name unseres ersten C-Programmes. Im Übrigen ist es völlig egal, ob Sie die Eingabe in Groß- und/oder Kleinbuchstaben eingeben, die Wirkung ist dieselbe.

Sollten Sie sich vertippt haben, kann das letzte Zeichen der Zeile mit der Backspace-Taste eliminiert werden. Diese Taste ist mit einem Pfeil nach links gekennzeichnet. Ansonsten haben Sie nur noch die Möglichkeit die gesamte Eingabezeile mit der Tastenkombination CTRL-X zu löschen.

Zum Abschluß jeder Zeile betätigen Sie die Return-Taste, die sich unterhalb der Backspace-Taste befindet. Nun erscheint ein leeres Fenster und der Text "Creating new file". Wir tippen nun unser Programm ein, wobei wir bei diesem Editor mittels der Cursortasten kreuz und quer über den Programmtext flitzen können, um gegebenenfalls Korrekturen und Änderungen vorzunehmen.

```
main()
{
 printf("Hallo, hier bin ich!");
}
```

Ach ja, falls Sie Probleme haben, die geschweiften Klammern auf der deutschen Tastatur zu finden: durch gleichzeitiges Drücken der folgenden Tasten erscheinen sie auf dem Monitor:

ALT(ernate), SHIFT und "Ü" = {
ALT(ernate), SHIFT und "+" = }

Welche Kombination Sie benutzen, ist Gewöhnungssache. Bei Verwendung der Tasten "4" und "5" des Zehnerblocks spart man sich jedoch die ALTernate-Taste.

Der Text wird jetzt in eine Datei unter dem Namen abgespeichert, den wir beim Aufruf von ED angegeben haben. In diesem Fall erhält das erste Programm den Namen "HALLO.C". Abspeichern läßt sich über die Tastensequenz "ESC", "S", "A". Wenn Sie den Editor auch gleich verlassen wollen, genügt ein "ESC" "X", wodurch die Datei abgespeichert und zum CLI zurückgekehrt wird. Mit "ESC" "Q" kann man ED auch ohne vorheriges Speichern beenden. Weitere Informationen zur Bedienung des Editors entnehmen Sie bitte Ihrem AMIGA-DOS-Handbuch (Kapitel 3 "ED").

Bedienen wir uns nun der Befehlsfolge "ESC" "X", und schon befinden wir uns wieder im CLI-Fenster. Damit hätten wir den Sourcecode, also unser C-Programm im Urzustand erzeugt.

3.2 Compiler im Einsatz

Jetzt rufen wir den Compiler auf, sei es über eine MAKE-Datei oder direkt per Tastatureingabe, und versorgen ihn mit dem Namen des zu übersetzenden C-Source. Da wir zu den bequemen Leuten gehören, verwenden wir die hierfür konzipierte MAKE-Datei. Das hat auch den Vorteil, das die compilerabhängigen Parameter und Optionen nicht für Verwirrung sorgen. Die benutzte MAKE-Datei (siehe Anhang) trägt, wie nicht anders zu erwarten, den Namen "MAKE". Deshalb geben Sie bitte untenstehende Zeile ein. (RETURN-Taste am Ende nicht vergessen!)

```
execute make hallo
```

(Je nachdem, wie die obige MAKE-Datei namens MAKE aussieht, braucht man die Endung ".C" nicht mitanzugeben! Die hier verwendete Version benötigt keine Extension.)

Auf dem Bildschirm sollte nun erscheinen (beim Lattice-C):

```
Compiling hallo.c
Lattice AMIGA 68000 C Compiler (Phase 1) V3.03
Copyright (C) 1984 Lattice, Inc.
Lattice AMIGA 68000 C Compiler (Phase 2) V3.03
Copyright (C) 1984 Lattice, Inc.
Module size P=0000001E D=00000015 U=00000000

Modules compiled: 1
-- Linking... hallo.o to hallo
Amiga Linker Version 2.20
Copyright (C) 1985 by Tenchstar Ltd., T/A Metacomco.
All rights reserved.
Linking complete - maximum code size = 11460 ($00002CC4) bytes
-- done compiling and linking 'hallo'. --
```

Na, hat alles wie gewünscht geklappt? Dann tippen Sie doch einmal "DIR" ein, um das Inhaltsverzeichnis der aktuellen Diskette zu erhalten. Dort befindet sich nun das erste ausführbare Programm unter dem Namen "HALLO". Man beachte, daß diese Datei keine Endung hat. Zudem entdeckt man auch noch Files, die die Endungen ".MAP", ".O" und natürlich ".C" aufweisen. Wie Sie sehen, ist die Kennung zur Unterscheidung unabdinglich. Aber nun wollen Sie Ihr erstes Programm bestimmt gleich ausprobieren. Rufen Sie es durch Eingabe der folgenden Zeile auf:

```
hallo
```

Auf dem Bildschirm erscheint der Text:

```
Hallo, hier bin ich!
```

Nicht gerade berauschend, was wir da geschafft haben, aber dies ist nur der Anfang. Was sollen wir aber machen, wenn der Compiler Fehler meldet?

3.3 Klappt es nicht?

Wenn Sie genau hingesehen haben, müßte Ihnen das Semikolon hinter der schließenden Klammer von printf aufgefallen sein. Dieses Semikolon ist eines der meist verwendeten Zeichen in C-Programmen, da es das Ende eines Befehls anzeigt. Daher steht hinter (fast) jeder C-Anweisung oder Funktion ein solches Semikolon. Sollten Sie es einmal vergessen, so hagelt es Fehlermeldungen von Ihrem Compiler. Nehmen wir an, Sie hätten unglücklicherweise das lebenswichtige Semikolon hinter dem "printf"-Aufruf übersehen. Der (Lattice-)C-Compiler meldet daraufhin:

```
hallo.c 4 Error 57: semi-colon expected
LC:LC1 failed returncode 10
```

Stellen wir uns mal ganz dumm und versuchen aus den Angaben des Compilers zu erfahren, was ihn an unserer Datei stört. Die erste Information ist die Datei, in der der Fehler aufgetaucht ist: "HALLO.C". Das ist gar nicht so banal, wie es auf den ersten Blick aussieht (werden wir später noch feststellen). Darauf folgt die Zeilennummer (4), Fehlernummer (57) und die Beschreibung des Fehlers im Klartext. Diejenigen unter Ihnen, die auch der englischen Sprache mächtig sind, sind hier eindeutig im Vorteil. Übersetzt bedeutet die Meldung, daß ein Semikolon in Zeile 4 erwartet wurde. Das trifft doch die Sache schon ziemlich gut.

Unsere Aufgabe ist es nun, den Editor wieder mit

```
ED HALLO.C
```

zu laden (diesmal wieder mit Endung) und in Zeile 4 zu gelangen. Sie können entweder die Zeilen abzählen, was bei dem Programm kein Problem darstellt und sicherlich die schnellste Methode ist, oder durch Tastensequenz "ESC" "M" "4" (Zeilennummer) diese Aufgabe dem Computer übertragen. Der Rechner stellt dann automatisch den Cursor in die angegebene Zeile.

Die 4. Zeile, die ja nur aus der geschweiften klammer besteht, ist aber völlig in Ordnung. Der wirkliche Fehler wurde schon in der vorherigen Zeile gemacht, da dort das Semikolon fehlt. Die Fehlermeldungen des C-Compilers darf man also nie auf die Goldwaage legen, sondern man muß die Suche großzügig in der weiteren Umgebung des Fehlers beginnen.

Nachdem Sie das Semikolon dahin gesetzt haben, wo es hingehört (hinter die printf-Funktion), speichern Sie die Datei wieder ab und beginnen wieder mit dem Aufruf der MAKE-Datei.

4. Theorie und Praxis

Nachdem Sie nun das "Wie" wissen, kommen wir zum "Warum".

Erklärung zum ersten Programm:

Die erste Zeile enthält den Funktionsnamen "main". Diese Funktion ist das wichtigste Element eines C-Programmes; ohne die main-Funktion läuft (im wahrsten Sinne des Wortes) gar nichts. Ein C-Programm besteht meist aus mehreren, manchmal bis zu hundert Funktionen. Eine Funktion löst deshalb immer nur eine Teilaufgabe des gesamten Programms. Sie ist durch das Paar runde Klammern als Funktion gekennzeichnet. Beginn und Ende der Funktion werden durch die geschweiften Klammern angezeigt. Innerhalb dieser Klammern stehen dann die Befehle, die der Rechner ausführen soll. Wenn Sie sich diese Regeln merken, können Sie auf einen Blick sagen, daß beispielsweise folgende Zeilen Funktionen aufrufen:

```
printf("Hallo");
warte(10);
musik();
ende();
```

und die untenstehenden Anweisungen wiederum keine solchen Routinen sind:

```
warte = alt;
geheim;
ende;
```

Wenn wir nun unserer compiliertes Programm mittels CLI starten, wird zuerst die main-Funktion angesprungen. Dies ist immer der Fall, egal, wo Sie diese Funktion im Listing plazieren. Sie kann am Anfang, mitten drin oder auch am Ende der Datei stehen, ausgeführt wird sie stets zuerst. In der Funktion main wird nun die Funktion printf aufgerufen. Diese Routine ist in einer der Bibliotheken untergebracht, so daß Sie nur zu wissen brauchen, wie sie heißt (printf), was sie macht (gibt einen Text auf den Bildschirm aus) und welche Informationen

sie benötigt (einen Text). Die Informationen, die der Funktion beim Aufruf übergeben werden, heißen deshalb auch Übergabeparameter oder Argumente. Die Übergabeparameter werden innerhalb der Klammern der aufzurufenden Funktion plaziert, damit die Funktion auch diese Werte geliefert bekommt. Wichtig ist hierbei, daß Sie den Text, eine Zeichenkette, mit "Gänsefüßchen" markieren.

Nachdem die printf-Funktion nun aufgerufen wurde, erscheint der Text auf dem Bildschirm. Die printf-Funktion hat ihre Arbeit erledigt, so daß es im Programm dort weitergeht, wo es durch den Funktionsaufruf unterbrochen wurde: hinter der printf-Anweisung.

Aber keine weiteren Anweisungen folgen denn der Zeile nach printf! Dies bedeutet, daß auch die main-Funktion an ihrem Ende angelangt ist und die Ausführung beendet, wodurch das Programm verlassen wird. Sie befinden sich wieder im CLI und können weitere Befehle eingeben. Das Ende des Programms und die daraus resultierende Rückkehr zum CLI stellt die Beendigung der main-Funktion dar. Sie merken, wie wichtig diese Funktion für uns ist, sie stellt das eigentliche C-Programm dar. Der Programmablauf beginnt mit der main-Funktion und endet mit ihr.

4.1 Die Formatierung des Programms

Kommen wir nun zur Formatierung, also dem Aufbau und der Strukturierung des Programms. Die Leerzeichen, Zeilenschaltungen und Absätze, die wir im Listing einfügen, interessieren den C-Compiler überhaupt nicht, sie werden von ihm ignoriert. Die gesamte Formatierung dient somit lediglich der Übersichtlichkeit eines Listings und hat auf den Programmablauf und die Länge des fertigen Programms keinerlei Einfluß. Daher könnte unser erstes Programm genauso gut wie eine der folgenden drei Versionen aussehen, die allesamt den gleichen Text ausgeben.

Version 2:

```
main(){
printf("Hallo, hier bin ich!");
}
```

Version 3:

```
main()
  {
  printf("Hallo, hier bin ich!");
  }
```

Version 4:

```
main(){printf("Hallo, hier bin ich!");}
```

Es bleibt Ihrem eigenen Geschmack überlassen, welche der vorgestellten Version Ihnen am übersichtlichsten erscheint. Wenn Sie sich jedoch einmal für einen Stil entschieden haben, so sollten Sie diesen kontinuierlich beibehalten. Besonders die letzte Version zeigt, wie unübersichtlich man bereits kleine Programme "strukturieren" kann. Chaos und Rätsel raten sind hier vorprogrammiert!

Übrigens, alle Befehle und Funktionen müssen in Kleinschrift eingegeben werden, da in C zwischen Groß- und Kleinschrift unterschieden wird. Sollten Sie anstelle von "printf" fälschlicherweise "Printf" oder "PRINTF" eintippen, so erhalten Sie vom Linker die Fehlermeldung, daß er diese Funktion nirgends vorfinden kann.

4.2 Definition einer Funktion

Eine Funktion führt immer nur das aus, was innerhalb der geschweiften Klammern eingetragen ist. Sollte dort nichts stehen, so macht der Rechner auch nichts. Ein Programm, das nichts tut, ist zwar nicht gerade aufregend, aber ein gutes Anschauungsbeispiel:

```
main()
{}
```

Erwarten Sie von diesem Programm keine Wunder. Wenn Sie das Programm compilieren, linken und starten, wird das Programm geladen und es passiert... nichts! Sie befinden sich schon wieder im CLI.

Dieses ist das kürzeste C-Programm, das es gibt (Wer eine kürzere Version schreiben kann, möge sie mir zusenden!).

Die Beschreibung, der in den Klammern eingeschlossenen Befehle für eine Funktion, nennt man Definition. Es wird festgelegt, was der Rechner unternehmen muß, wenn er den Auftrag erhält, eine bestimmte Funktion auszuführen. Solche Fremdworte werden Sie bestimmt noch häufiger antreffen, so daß es sinnvoll ist, sie sich jetzt schon einzuprägen.

Zurück zur Praxis: Eine Funktion kann natürlich auch mehrere Befehle oder Funktionsaufrufe beinhalten. So läßt sich ein größerer Text beispielsweise mit diesem Programm auf den Bildschirm bringen:

```
main()
{
  printf("Hallo, ich habe da eine Frage!\n");
  printf("Glauben Sie an ein Leben ohne Strom?\n");
  printf("Ich nicht!\n");
}
```

Die Ausgabe auf Ihrem Monitor wird so aussehen:

```
Hallo, ich habe da eine Frage!
Glauben Sie an ein Leben ohne Strom?
Ich nicht!
```

Neu in diesem Programm ist nicht nur die Anzahl der printf-Aufrufe (jetzt 3 statt 1), sondern auch das Steuerzeichen '\n', welches sich innerhalb der Anführungsstriche befindet. Die einzelnen Funktionsaufrufe können beliebig aneinandergehängt werden, lediglich auf das abschließende Semikolon ist unbedingt zu achten.

4.3 printf und Steuerzeichen

Der Text wird so, wie er zwischen den Anführungsstrichen plaziert ist (auch mit Leerzeichen), auf den Bildschirm geschrieben. Ausnahme bilden nur die Steuerzeichen, von denen hier der erste Vertreter auftritt: '\n'. Ein Steuerzeichen können Sie an dem vorangestellten Backslash '\' erkennen, das folgende Zeichen gibt dann an, welche Funktion ausgeführt werden soll. Das 'n' bedeutet hierbei, daß ein Zeilenvorschub durchgeführt wird, zu deutsch, der Rechner schreibt den nächsten Text an den Anfang der folgenden Zeile. Der Text erscheint also nicht automatisch bei jedem neuen Aufruf der printf-Funktion auch in einer neuen Zeile, dafür müssen Sie schon selbst sorgen. Dies haben Sie ja auch daran gemerkt, daß das Prompt, welches nach Beendigung unseres ersten Programmes erschien, direkt hinter unseren Text ausgegeben wurde.

Grundsätzlich werden alle Zeichen hintereinander auf den Bildschirm geschrieben, auch wenn sie durch verschiedene Funktionsaufrufe produziert werden. Wenn der Text in eine neue Zeile geschrieben werden soll, muß dies durch das Steuerzeichen '\n' dem Rechner mitgeteilt werden. Wie aus obigem Beispiel zu erkennen ist, ist dies dreimal der Fall. Dabei sind Sie nicht daran gebunden, dieses Steuerzeichen jedesmal am Ende einer Zeichenkette zu verwenden. Es kann mitten zwischen den anderen "normalen" Zeichen oder auch am Anfang stehen. Wenn Sie

wollen, könnten Sie alle drei Zeilen mit einem printf-Aufruf "verarzten":

```
printf("Hallo, ich habe da eine Frage!\nGlauben Sie an ein Leben
ohne Strom?\nIch nicht!\n");
```

Wie Sie sehen ist diese Zeile sehr unübersichtlich, weshalb wir uns auch nicht weiter damit befassen. Wenn Sie sich generell bei Verwendung des '\n"-Zeichen angewöhnen, dieses am Ende eines Strings zu benutzen (so wird eine Zeichenkette auch bezeichnet), werden Sie keine Verständnisprobleme bekommen.

Das Steuerzeichen '\n' ist nicht die einzige Möglichkeit, den Text aufzubereiten, falls man diesen Ausdruck überhaupt auf eine simple Zeilenschaltung anwenden darf. Der Name "printf" besteht zum einen aus dem englischen Wort print, was soviel wie schreibe, drucke bedeutet. Diese Funktion ist beispielsweise mit dem BASIC-Befehl PRINT oder der Pascal-Anweisung "write" vergleichbar. Der zweite Teil, nämlich das "f", gibt einen Hinweis darauf, daß wir den Text formatiert ausgeben können. Diese Funktion ist also in der Lage, Zeichenketten und andere Übergabewerte nach allen Wünschen zu bearbeiten und auszugeben. Da diese Möglichkeiten sehr vielfältig sind, möchte ich Sie nicht gleich damit "erschlagen". Wir werden sie nacheinander besprechen, und zwar dann, wenn sie gerade benötigt werden.

4.4 Kommentare

In C ist es möglich Kommentare einzufügen, die es dem Programmierer ermöglichen, zusätzliche Informationen im Listing zu plazieren, die aber keinerlei Auswirkungen auf den Programmablauf haben. Das ist genauso wie bei der Formatierung des Programms, sie haben keinerlei Auswirkungen auf Geschwindigkeit oder Größe des resultierenden Programmes. Kommentare beginnen mit den beiden Zeichen "/*" und enden mit "*/". Der Compiler überliest alles, was innerhalb dieser Begrenzungen steht. Es besteht also keine Notwendigkeit, mit Kommentaren zu sparen, da sie nirgends im endgültigen Programm auftauchen.

```
main()
{
  /* Dieses Programm gibt einen Text aus, */

  printf("---Kommentare---erwünscht---stop---\n");
  /* der /hier beginnt, und dort hinten endet! / */
}
```

Das Repertoire, das uns bis jetzt zur Verfügung steht, ist ja noch recht bescheiden. Wir können Programme schreiben, die beliebig lange Texte auf dem Bildschirm ausgeben und diese gegebenenfalls mit Bemerkungen versehen.

Es wäre natürlich nicht schlecht, wenn wir dem Rechner auf eine von ihm gestellte Frage auch eine Antwort geben könnten. Hierfür benötigen wir eine Funktion, die Daten einlesen kann, somit also genau das Gegenteil der printf-Funktion darstellt. Auch an eine solche Funktion wurde gedacht, sie heißt scanf.

Bevor wir allerdings blindlings in unser Unglück stürzen, sollten wir uns vielleicht erst einmal ein paar Gedanken über den Programmablauf machen. Das Programm soll einen Text, genauer eine Frage auf dem Bildschirm bringen. Soweit kein Problem, hierfür benutzen wir die printf-Funktion. Dann soll die Antwort auf die Frage eingelesen werden. Nur: Um welche Gruppe von Daten handelt es sich dabei?

Um die Aufgabe etwas zu vereinfachen, nehmen wir an, es soll eine Kennzahl als Antwort eingegeben werden. Auf die Fragestellung

```
Geht es Ihnen gut?
(1) = JA, (2) = NEIN

Kennzahl:
```

soll eine 1 oder eine 2 eingetippt werden. Diese Zahlen müssen ja auch irgendwo hinterlegt oder abgespeichert werden. Zu diesem Zweck stellt uns C eine Reihe von Variablentypen bereit, welche wir nun benutzen können.

4.5 Erste Bekanntschaft mit Variablen und Arithmetik in C

Variablen sind Speicher für bestimmte Gruppen von Informationen, je nachdem, welche Werte sie aufnehmen sollen. Es gibt Variablen für Zeichen und Zeichenketten, für verschiedene Arten und Größen von Zahlen und Kombinationen von beiden. Eine Variablengruppe kann ganze Zahlen darstellen, sogenannte Integerwerte. Wie ein Programm aussieht, das solche Variablen benutzt, zeigt das folgende Beispiel:

```
main()
{
  int eingabe;

  printf("Geht es Ihnen gut?\n");
  printf("(1) = JA, (2) = NEIN\n\n");
  printf("Kennzahl: ");
  scanf("%d", &eingabe);
  printf("\n\nIhre Eingabe lautete %d\n", eingabe);
}
```

Auf den ersten Blick sieht das ja ziemlich wild aus. Gleich die erste Zeile innerhalb von main dürfte uns Kopfzerbrechen bereiten. Hier wurde eine Variable definiert. Ähnlich wie bei der Funktionsdefinition zeigen wir dem Compiler, was wir mit dieser Variablen anstellen wollen.

4.5.1 Integer

Das erste Wort "int" beschreibt den Variablentyp. Von ihm hängt es ab, was wir in einer Variablen hinterlegen können. Bei "int", einem Integerwert, lassen sich alle ganzen Zahlen zwischen (ca.) -32000 und +32000 abspeichern. Für unsere bescheidenen Werte 1 und 2 reicht also dieser Typ voll aus.

Die Variable erhält nun im folgenden einen Namen zugewiesen, über den wir stets auf den in ihr abgelegten Wert zugreifen können. Hier wurde sie "eingabe" getauft, was auch ihrer Auf-

gabe gerecht wird. Die Definition einer Variablen wird mit dem Semikolon abgeschlossen.

In den folgenden Zeilen entdecken wir nur die bereits vertrauten printf-Aufrufe. Zur Eingabe benutzen wir die schon angesprochene scanf-Funktion, die ähnlich komplex ist wie printf. Da es ja eine Unzahl verschiedener Datentypen gibt, müssen wir der Eingaberoutine auch mitteilen, welche Daten eigentlich erwartet werden. Dies geschieht über die Formatanweisung "%d". Das Prozentzeichen charakterisiert Formatanweisungen, gefolgt von dem Zeichen, das die gewünschte Funktion angibt. Auch hier sind Parallelen zu den Steuerzeichen, die ebenfalls von einem bestimmten Zeichen, dem Backslash ('\'), eingeleitet werden. Das darauf folgende "d" teilt scanf mit, daß der einzulesende Wert vom Typ "int" sein soll. Dieser Zeichenkette wird nun die gewünschte Variable durch ein Komma getrennt hinten angehängt. Die Besonderheit bei dieser Funktion ist das vorangestellte Kaufmanns-Und "&". Es ist wichtig, daß Sie sich diese Besonderheit merken. Spätestens dann, wenn der Rechner abstürzt, sollte man vielleicht einmal einen Blick auf die Parameter der scanf-Funktion werfen.

Ist scanf zur vollen Zufriedenheit mit Informationen versorgt, kann der Benutzer die Eingabe tätigen. Er gibt eine Zahl ein und betätigt anschließend die Return-Taste. Dieser Wert findet sich nun in der Variablen "eingabe" wieder. Das Programm kann diese Zahl nun auswerten. Wir beschränken uns aber im Moment auf die Bestätigung der Eingabe. Hierfür wird, wie sollte es anders sein, wieder die printf-Funktion benutzt, deren Domäne bekanntlich Ausgaben aller Art sind. Auch diese Funktion muß natürlich mitgeteilt bekommen, welche Art von Daten sie verarbeiten soll. Hierfür wird wieder eine Formatanweisung benötigt, die glücklicherweise identisch mit der in scanf benutzten ist. printf weiß dadurch, daß eine Integerzahl übergeben wird, die an der Stelle in den Text eingefügt werden soll, an der auch die Formatanweisung steht. Die entsprechende Variable, hier "eingabe", folgt diesem String durch ein Komma getrennt.

Und schon erscheint die getätigte Eingabe wieder auf dem Monitor. Das ist aber nur dann der Fall, wenn Sie nicht auf "dumme" Ideen kommen und anstelle einer 1 oder 2 vielleicht Text eingaben, der hier ja gar nichts zu suchen hat. In diesem Falle lautet die Eingabe plötzlich 65535 (Lattice-C), obwohl diese Zahl nirgends in der Eingabezeile auftaucht. Wenn Sie allerdings versuchen ohne Eingabe zu entkommen, so bleibt scanf hartnäckig. Der Bildschirm wird zwar durch ein Leerzeile hochgescrollt, aber es wird erneut eine (vernünftige) Eingabe erwartet.

4.5.2 Wenn, dann... - der if-Befehl

Es ist recht eintönig, den Computer nur die Eingabe "nachplappern" zu lassen. Besser wäre doch eine Reaktion auf die Eingabe in der Weise, daß er bei einer '1' den Text ausgibt "Das ist erfreulich!", ansonsten "Schade, das tut mir leid!". Der Rechner müßte also in der Lage sein, den Wert, der in "eingabe" steht, mit anderen Zahlen zu vergleichen. Je nachdem wie dieser Test ausgefallen ist, muß er den einen oder den anderen Text ausgeben. Im Deutschen würde man eine solche Konstruktion *wenn... dann...* nennen. Ins Englische übersetzt hieße das dann *if... then...* Wie in vielen anderen Sprachen auch, finden wir den "if"-Befehl auch in C wieder. Das "then" können wir uns schenken, es wird in C nicht benötigt.

Nun wollen wir die Variable mit 1 vergleichen. Ist diese Bedingung zutreffend, also wahr, so wird die dem if-Befehl folgende Anweisung ausgeführt. Am besten, wir nehmen uns diesen Abschnitt einmal vor:

```
if(eingabe == 1)
  printf("Das ist erfreulich!\n");
if(eingabe == 2)
  printf("Schade, das tut mir leid!\n");
```

Die Bedingung wird in runde Klammern gefaßt, so daß der erste printf-Aufruf nur dann erfolgt, wenn "eingabe" gleich 1 ist. Das gleiche gilt auch für den folgenden if-Befehl, mit dem Unterschied, daß hier auf 2 geprüft wird. Schauen Sie genau hin!

Hinter dem if steht nie ein Semikolon. Wenn Sie experimentierfreudig sind, können Sie ja mal ausprobieren, was passiert, wenn Sie z.B. hinter das zweite if ein ';' setzen.

Ich sag es Ihnen besser gleich: Das Programm verhält sich so, als wäre die Zeile "if(eingabe == 2);" gar nicht vorhanden. Nach jeder Eingabe erscheint der Satz "Schade, das tut mir leid!". Hier heißt es Augen aufhalten!

Mit den obigen 4 Zeilen läuft unser Programm zwar schon einwandfrei, aber wir können es noch verbessern. Aus BASIC kennen Sie vielleicht die Sequenz:

```
IF A=1 THEN PRINT "Schön für Sie!": ELSE PRINT "Pech gehabt!"
```

Auch C bietet ein "else", das nur zusammen mit if verwendet werden kann. "else" dient dazu, einen Befehl dann auszuführen, wenn die Bedingung nicht erfüllt ist. Dadurch können wir uns die zweite Abfrage schenken.

```
if(eingabe == 1)
  printf("Das ist erfreulich!\n");
else
  printf("Schade, das tut mir leid!\n");
```

Das C-Listing sind nach den vorgenommenen Änderungen nun so aus:

```
main()
{
  int eingabe;

  printf("Geht es Ihnen gut?\n");
  printf("(1) = JA, (2) = NEIN\n\n");
  printf("Kennzahl: ");
  scanf("%d", &eingabe);
  if(eingabe == 1)
    printf("Das ist erfreulich!\n");
  else
    printf("Schade, das tut mir leid!\n");
}
```

Soweit alles klar? Gut, wenn Sie keine Probleme haben, dann machen wir uns welche. Wenn Sie nach dem if mehrere Befehle, z.B. zwei printf-Aufrufe, ausführen wollen, so kann man nicht die zweite Zeile einfach dahinter schreiben.

```
if(eingabe == 1)
  printf("Das ist erfreulich!\n");
  printf("Hoffentlich bleibt es so!\n"); /* So nicht ! */
else
  printf("Schade, das tut mir leid!\n");
```

Da stets nur ein Befehl nach if ausgeführt wird, müssen wir uns etwas anderes einfallen lassen. Bislang haben wir nur von Anweisung gesprochen, korrekter wäre aber Anweisungsblock. Setzen wir nämlich mehrere Befehle in geschweifte Klammern, so haben wir einen Anweisungsblock, der als Gesamtheit gilt. Diesen Block können wir dann ohne Probleme hinter die if-Abfrage setzen.

```
if(eingabe == 1)
  {
    printf("Das ist erfreulich!\n"); /* Richtig! */
    printf("Hoffentlich bleibt es so!\n");
  }
else
  printf("Schade, das tut mir leid!\n");
```

So weit, so gut. Der AMIGA kann eine ganze Menge, was er aber am besten kann, ist rechnen (und das wahnsinnig schnell!). Und genau das werden wir nun in C versuchen. Beginnen wir mit der Addition. Um zwei Werte zu addieren, werden sie mit dem Pluszeichen '+' verknüpft.

```
summe = zahl1 + zahl2;
```

Das Ergebnis wird im obigem Beispiel in der Variablen "summe" abgelegt. Diese Variablen müssen wie alle Variablen am Anfang der Funktion erst definiert werden. Benutzen wir wieder den Typ Integer, so benutzen wir folgende Definition:

```
int summe;
int zahl1;
int zahl2;
```

Falls Sie es noch nicht wissen sollten, C ist ein Sprache für Schreibfaule. Es gibt so gut wie für alles irgendwelche Abkürzungen, so daß die Tastatur nicht allzu stark strapaziert wird. Und wer ein guter C-Programmierer werden will, sollte von diesem Komfort Gebrauch machen. In der Definition der drei Variablen haben wir jedesmal den gleichen Datentyp vorliegen. Deshalb brauchen wir "int" nur einmal hinzuschreiben und alle Variablen, die Integerwerte sein sollen dahinter aufzureihen.

```
int summe, zahl1, zahl2;
```

Alle Variablen werden durch Kommata getrennt, die Liste wird mit einem Semikolon abgeschlossen.

Nach diesen Informationen ist es uns nun möglich, ein Additionsprogramm für zwei Zahlen zu schreiben. Die Dateneingabe obliegt wieder der scanf-Funktion, wobei wir diesmal zwei Zahlen einlesen wollen. Dies machen wir nicht etwa durch zwei separate Funktionsaufrufe (das geht natürlich auch!), sondern wir schreiben ein zweites Formatzeichen in den ersten String der scanf-Routine. Dieser enthält unmittelbar hintereinander "%d%d" (ohne Leerzeichen), wodurch wir veranlaßt werden, auch noch den zweiten Parameter anzugeben. Und hier das Listing:

```
main()
{
  int summe, zahl1, zahl2;

  printf("Bitte geben Sie zwei Zahlen ein!\n");
  scanf("%d%d",&zahl1, &zahl2);
  summe = zahl1 + zahl2;
  printf("%d + %d = %d\n", zahl1, zahl2, summe);
}
```

Sollten Sie bereits etwas Erfahrung in einer anderen Programmiersprache beispielsweise BASIC haben, so können Sie die beiden Listings miteinander vergleichen. Besonders der letzte

printf-Aufruf mit drei im Text verstreuten Formatanweisungen mag unübersichtlich erscheinen.

```
10 PRINT "Bitte geben Sie zwei Zahlen ein!"
20 INPUT Z1,Z2
30 SU = Z1 + Z2
40 PRINT Z1;" +";Z2;" =";SU
```

4.5.3 Rechnen mit C

Raten Sie mal, was verändert werden müßte, um anstelle der Addition eine Multiplikation durchzuführen. Das Pluszeichen wird durch das Sternchen '*' ersetzt. Für die Subtraktion wird '-' verwendet und für die Division der Schrägstrich '/'.

Man kann aber nicht nur mit Variablen, sondern auch mit Konstanten rechnen. Beide Gruppen können auch gemischt werden. Einige Beispiele für korrekte Berechnungen zeigt die folgende Liste. Es wird dabei vorausgesetzt, daß alle Variablen definiert sind und einen "sinnvollen" Wert beinhalten.

```
ergebnis = zahl * 4;
summe = var + 2 + var2 + 3 + var4;
ergebnis = 4 * 5 - 7 / var;
wert = 2 * (zahl - 7);
result = 4 + 5 * 3 - 2;
zaehler = zaehler + 1;
```

Eine Formel zu berechnen, ist also genauso einfach, als wenn Sie diese in einen Taschenrechner eintippen. C kennt nämlich auch die mathematischen Gesetze, wie z.B. Punkt-vor-Strich-Regel und Klammerregeln. Im vorletzten Beispiel erhält man als Ergebnis 17 und nicht 25, da zuerst das Produkt aus 5 * 3 berechnet und dann erst 4 addiert und 2 subtrahiert wird.

Die letzte Zeile verdient ein besonderes Augenmerk. Diese Gleichung ist im mathematischen Sinne absoluter Blödsinn, sie ist unlösbar. Für den Computer stellt sich aber kein Problem. Er nimmt den Inhalt der Variablen "zaehler", addiert eins hinzu und

speichert das Resultat wieder in "zaehler". Durch diese Operation zählt man bei jedem Aufruf dieser Zeile die Variable um eins hoch.

Eine Bemerkung hierzu: Wie Sie an den Beispielen gesehen haben, benutzt man für die Zuweisung einer Zahl an eine Variable immer ein Gleichheitszeichen "=". Für Vergleiche, wie beispielsweise bei einer if-Abfrage, werden aber weiterhin immer zwei Gleichheitszeichen "==" unmittelbar hintereinander verwendet.

Wollen wir unsere ersten zaghaften Versuche im Bereich der Algebra durch ein etwas umfangreicheres Programm stützen. Die Anforderungen daran sind, daß alle Grundrechenarten mit zwei Variablen durchgeführt werden sollen. Die zwei Zahlen entgegenzunehmen, ist für uns bereits Routinearbeit, der Operator wird mittels Kennzahl eingelesen. Anschließend fragt man diese Zahl mit mehreren if-Anweisungen ab, ob diese oder jene mathematische Operation ausgeführt werden soll. Steht fest, daß eine bestimmte Operation gewünscht ist, so wird das Ergebnis entsprechend berechnet. Sollte keine gültige Kennzahl eingegeben worden sein, so erscheint ein entsprechender Hinweis.

Betrachten Sie nun zunächst einmal das folgende Listing, das die gestellten Aufgaben bereits fehlerfrei löst:

```
main()
{
  int zahl1, zahl2, ergebnis, operator, fehler;

  printf("Bitte geben Sie zwei Zahlen ein!\n");
  scanf("%d%d", &zahl1, &zahl2);
  printf("Gut, und nun die Kennzahl fur die Operation:\n");
  printf("1=Addition, 2=Subtraktion, 3=Multiplikation, 4=Division\n");
  scanf("%d", &operator);
  fehler = 1;
  if(operator == 1) /* Addition */
    {
      ergebnis = zahl1 + zahl2;
      fehler = 0;
    }
```

```
  if(operator == 2) /* Subtraktion */
    {
       ergebnis = zahl1 - zahl2;
       fehler = 0;
    }
  if(operator == 3) /* Multiplikation */
    {
       ergebnis = zahl1 * zahl2;
       fehler = 0;
    }
  if(operator == 4) /* Division */
    {
       ergebnis = zahl1 / zahl2;
       fehler = 0;
    }
  if(fehler == 1) /* Keine der obigen Bedingungen erfuellt? */
    printf("Falscher Operator! Nur Ziffern 1-4 eingeben!\n");
  else
    printf("Das Ergebnis lautet %d\n", ergebnis);
}
```

Die Verarbeitung ist sehr übersichtlich: Nachdem alle Werte eingegeben worden sind, wird der Variablen "fehler" der Wert eins zugewiesen. Bei jeder nun folgenden Operation, sei es Addition, Subtraktion oder dergleichen, wird die Variable wieder auf Null gesetzt. So ist festzustellen, ob eine der vier Berechnungen durchgeführt wurde. Im anderen Fall mußte der Wert in "operator" einen Wert enthalten, der nicht erlaubt wurde.

Deshalb wird vor der Ausgabe des Ergebnisses überprüft, ob es überhaupt berechnet wurde, was an der Variable "fehler" zu erkennen ist.

Testen Sie das Programm mit verschiedensten Werten gründlich durch, beachten Sie aber, daß ja in unseren Variablen nur Zahlen zwischen +32000 und -32000 gespeichert werden können. Außerden sollten Sie sich hüten, durch 0 zu dividieren. Dieser Wert wird von unserem Programm nicht abgefragt, so daß eine solche Aufgabe unweigerlich einen Systemabsturz inclusive Guru-Meldung zur Folge hat.

Ist Ihnen etwas aufgefallen? Nein, ich gebe Ihnen einen Tip: Versuchen Sie es doch mal mit der Division von 9 durch 2. Als Ergebnis gibt der Rechner 4 aus, was aber nicht stimmt (4,5 wäre richtig!). Ist dieser sündhaft teure Supercomputer nicht einmal in der Lage, richtig zu dividieren?

4.5.4 Fließkommazahlen

Nur keine Aufregung, dieser kleine Fehler ist nicht auf den AMIGA zurückzuführen, sondern auf unsere Variablenart. Wenn Sie sich erinnern, sind int-Variablen ja nur imstande ganze Zahlen zwischen ±32000 zu verarbeiten. Der Wert 4,5 ist aber keine ganze Zahl, sondern eine "Kommazahl", ein sogenannter Dezimalbruch. Wenn also irgendwo bei einer Division ein Rest, hier ein Nachkommateil auftaucht, so fällt dieser ganz dezent unter den Tisch. Das bedeutet aber nicht, daß wir das Ergebnis aus 9/2 nicht mit dem AMIGA berechnen könnten. Das einzige, was dazu benötigt wird, ist ein Variablentyp der auch Kommazahlen abspeichern kann. Kein Problem, auch hierfür ist C ausgerüstet! Diese Variablenart hört auf den Namen "float", was eine engl. Abkürzung für den Begriff Fließkommazahl ist.

Wollen wir unser bisheriges Programm auf diesen neuen Datentyp umstellen, so muß zumindest für unsere Variablen "zahl1", "zahl2" und "ergebnis" der Datentyp in "float" geändert werden. Dieser Vorgang besteht lediglich aus dem Austausch von "int" durch "float". Die ersten Zeilen sehen dann so aus:

```
main()
{
  float zahl1, zahl2, ergebnis;
  int operator, fehler;
```

Damit ist es aber noch nicht ganz getan, denn wir teilen der scanf und der printf mit der Formatanweisung "%d" mit, daß sie ein Integerwert zu erwarten haben. Das ist ja nun nicht mehr der Fall, so daß an die Stelle des "%d" ein "%f" tritt. Das "f" steht für Floatwerte, die übergeben werden sollen, genauso wie

das "d" für alle Integerwerte benutzt wird. Der erste scanf-Aufruf sieht dann so aus:

```
scanf("%f%f", &zahl1, &zahl2);
```

Das einzige, was noch fehlt, ist die entsprechende Änderung bei der letzten printf-Funktion. Dort wird die Variable "ergebnis" verwendet, die nun ja ebenfalls ein Floatwert ist. Auch hier verschwindet das "d" zugunsten eines "f". Neu compiliert und gelinkt ist mit dieser Version auch eine einwandfreie Berechnung von Kommazahlen möglich. Ein weiterer Vorteil kommt hinzu: Die Zahlen, die im Typ "float" gespeichert werden können, sind praktisch in ihrer Größe unbeschränkt, so daß selbst noch Millionen- und Milliardenbeträge berechnet werden können.

Neben "int" und "float", mit denen Zahlen aller Art abgespeichert werden können, benötigen wir noch eine Variablengattung die Zeichen aufnehmen kann. Es wäre im obigen Programm doch viel besser, man würde das Pluszeichen anstelle der Kennziffer 1 eintippen. Auch hieran wurde gedacht! Mit dem Datentyp "char" sind Variablen zu definieren, die Zeichen aufnehmen sollen.

4.5.5 Zeichen und Zeichenketten

Die Definition läuft genauso ab, wie schon bei float- und int-Variablen beschrieben:

```
char zeichen;
```

Auch diese Sorte von Variablen besitzt wieder eigene Formatzeichen für die Funktionen printf und scanf. Hier findet das "c" für "char" Verwendung. Um unserem Rechenprogramm noch den letzten Schliff zu geben, bauen wir nun noch das Einlesen eines Zeichens, dem Operator, ein. Dadurch entfällt die Eingabe einer Kennzahl. Noch eine kleine Verbesserung kann an dieser Stelle vorgenommen werden:

Die Eingabe soll wie beim Taschenrechner erfolgen können, also in der Reihenfolge: erste Zahl, Operator, zweite Zahl. Anstelle der Gleichheitstaste drücken wir auf Return oder Enter. Da die scanf-Routine so flexibel ist, genügt die folgende Umstellung, um diesen Wunsch zu erfüllen:

```
scanf("%f%c%f", &zahl1, &operator, &zahl2);
```

Wir haben einfach zwischen die beiden Formatanweisungen für Zahleneingabe das "%c" eingefügt und auch die folgenden Variablen in dieser Reihenfolge geordnet.

Die Abfragen, die zuvor die Kennzahl testeten, müssen nun Zeichen prüfen. Nichts leichter als dies! Das Zeichen braucht bloß in einfache "Gänsefüßchen" gesetzt zu werden.

```
if(operator == '+')
  ...
```

Der AMIGA hat auf seiner deutschen Tastatur zwar viele beschriftete Tasten, aber das für uns wichtige einfache Hochkomma ist nicht zu entdecken. Auch hier habe ich mir die Mühe gemacht, die richtige Tastenkombination zu erforschen: SHIFT + "#".

Die Taste mit dem Doppelkreuz "#" befindet sich links neben der Return-Taste. Alle anderen, dem einfachen Hochkomma ähnlichen Zeichen sollten Sie nicht verwenden, da sie lediglich zu einer Reihe von Fehlermeldungen führen. Unser "Gänsefüßchen" zeichnet sich durch einen kleinen Knick nach links aus, wodurch es von anderen Apostroph-Zeichen zu unterscheiden ist.

Nach all den kleinen Änderungen können Sie Ihre Version mit dem endgültigen Programm vergleichen, in dem ohne große Erläuterungen noch ein paar kosmetische Korrekturen vorgenommen wurden. Sie sollten jetzt bereits imstande sein, die bei printf vorgenommenen Ergänzungen zu verstehen.

Bei der Eingabe hat sich auch eine Kleinigkeit geändert. Bisher konnten Sie (brauchten aber nicht) nach jeder Zahl Return betätigen, jetzt muß die gesamte Eingabe in einer Zeile stehen. Der Grund liegt darin, daß wir ein einzelnes Zeichen mit "%c" einlesen wollen, was ja genausogut ein Return oder ein Leerzeichen sein kann. Deshalb folgt der ersten Zahl unmittelbar der Operator, dann kann, wenn Sie unbedingt wollen, auch wieder die Return-Taste benutzt werden. Zuletzt steht wieder die zweite Zahl, wie in dieser Zeile:

```
15.5*12.5 (RETURN)
```

Das Dezimalkomma wird durch einen Punkt ersetzt, ansonsten wird der eingetippte Dezimalbruch nicht akzeptiert. Als Ausgabe auf dem Bildschirm erhalten Sie nach der vorgeschlagenen Zeile:

```
15.500000 * 12.500000 = 193.750000
```

Hier jetzt die absolut letzte Version dieses Programms:

```
main()
{
  float zahl1, zahl2, ergebnis;
  char operator;
  int fehler;

  printf("Eingabeformat: Zahl Operator Zahl (ohne Leerzeichen)!\n");
  scanf("%f%c%f", &zahl1, &operator, &zahl2);
  fehler = 1;
  if(operator == '+') /* Addition */
    {
       ergebnis = zahl1 + zahl2;
       fehler = 0;
    }
  if(operator == '-') /* Subtraktion */
    {
       ergebnis = zahl1 - zahl2;
       fehler = 0;
    }
  if(operator == '*') /* Multiplikation */
    {
       ergebnis = zahl1 * zahl2;
       fehler = 0;
```

```
    }
  if(operator == '/') /* Division */
    {
       ergebnis = zahl1 / zahl2;
       fehler = 0;
    }
  if(fehler == 1) /* Keine der obigen Bedingungen erfuellt? */
    printf("Falscher Operator %c!\n", operator);
  else
    printf("%f %c %f = %f\n", zahl1, operator, zahl2, ergebnis);
}
```

Anmerkung: Sollten Sie mit dem Aztek arbeiten, müssen Sie außer der Standard-Library "c.lib" auch die Library für Mathematische Funktionen und Fließkommazahlen hinzulinken. Beispielsweise so:

```
ln grundr-fertig.o -lm -lc
```

Bleiben wir noch etwas bei den char-Variablen. Sie wissen ja, daß eine Zuweisung an einen solchen Datentyp etwa wie folgt lautet:

```
char zeichen;
zeichen = 'a';
```

Dadurch ist das Zeichen 'a' in der Variablen "zeichen" gespeichert. Besonders wichtig sind die einfachen Anführungsstriche, die den Buchstaben umschließen. Sollten Sie die einfachen mit den doppelten Anführungszeichen verwechseln, werden Sie Schwierigkeiten bekommen. Merken Sie sich also:

```
char zeichen;
zeichen = "x";
```

ist falsch! Es muß heißen:

```
char zeichen;
zeichen = 'x';
```

Ein Zeichen kommt selten allein, das ist nicht nur ein dummer Spruch, sondern Praxis. In vielen Fällen ist man gezwungen, nicht nur ein einzelnes Zeichen, sondern gleich eine ganze Zeichenkette entgegenzunehmen. Ein anderer Ausdruck für Zeichenkette ist z.B. String, der vor allem BASIC-Programmierern ein Begriff sein wird. In C stellen Strings aber keinen neuen Datentyp dar, sondern entsprechen vielmehr einer Menge von char-Werten. So sieht dann auch die Definition eines Strings aus:

```
char string[81];
```

Hierdurch wird dem Compiler mitgeteilt, daß der String 81 Einträge haben soll. Sollen längere Texte dort abgespeichert werden, so muß die Zahl innerhalb der eckigen Klammern entsprechend vergrößert werden. Die eckigen Klammern sind über

ALT + "ü" = [
ALT + "+" =]

zu erreichen. Jetzt können wir mal versuchen, einen Text über die Tastatur ein- und wieder auszugeben. Das Formatzeichen für Strings ist übrigens "%s", welches in scanf und printf verwendet werden kann.

Um die Verwendung von Strings zu zeigen, genügt nach den vorangegangenen Programmen ein relativ kurzes Listing. Das Programm fragt nach dem Namen und gibt ihn danach wieder auf dem Bildschirm aus.

```
main()
{
  char string[81];

  printf("Wie hei3en Sie denn mit Vornamen?\n");
  scanf("%s", string);
  printf("Hallo %s, darf ich Dich duzen?\n", string);
}
```

Sollten Sie beim Aufruf der scanf-Funktion etwas stutzen, dann haben Sie gut aufgepaßt. Dort fehlt nach unserem Wissen das "&"-Zeichen. In diesem Fall, beim Einlesen einer Zeichenkette,

ist dies gerade nicht erforderlich, mehr noch, es ist ganz einfach falsch. Warum das so ist, erfahren Sie im zweiten Teil, denn das ist nicht mehr ganz so einfach.

5. Schleifen und was dazu gehört

Die bisher vorgestellten Programme liefen stets geradlinig von oben nach unten ab. Man konnte zwar mittels einer if-Abfrage einige Teile überspringen, zurück zu weiter vorne liegenden Befehle konnte man sich aber nicht bewegen. Möchte man nun, daß bestimmte Teile mehrmals durchlaufen werden, muß man eine Schleife konstruieren.

5.1 while-Schleifen

Ein Vertreter dafür ist die while-Schleife. Wie dem if-Befehl folgt auch der while-Anweisung ein Paar runde Klammern, zwischen denen die Bedingung untergebracht ist. Ein Beispiel macht dies deutlich:

```
main()
{
  int zaehler;

  zaehler = 15;
  while(zaehler > 0)
    {
       printf("zaehler ist %d\n", zaehler);
       zaehler = zaehler - 1;
    }
}
```

Der Block, der nach "while" steht, wird solange ausgeführt, wie die Bedingung innerhalb der Klammern erfüllt ist. Zu Beginn wird der Variablen "zaehler" 15 zugewiesen. Solange die Bedingung "zaehler > 0" erfüllt ist, wird der folgende Block ausgeführt, der zum einen den augenblicklichen Wert von "zaehler" ausgibt, zum anderen ihn anschließend um eins verringert. In diesem Fall wird 15mal die printf-Funktion aufgerufen, ehe "zaehler" auf 0 heruntergezählt ist. Außer dem bekannten Vergleich mittels "==" ist auch ein Test auf größer ">", kleiner "<", größer oder gleich ">=" oder auch kleiner gleich "<=" möglich. Wir benutzen hier den Vergleich, ob "zaehler" größer als 0 ist.

Die Vergleichsoperatoren finden Sie so gut wie in jeder Programmiersprache wieder.

<	kleiner als
<=	kleiner oder gleich
>	größer als
>=	größer oder gleich
==	gleich
!=	ungleich

Anstelle von

```
while(zaehler > 0)
```

könnten Sie genauso gut

```
while(zahler >= 1)
```

schreiben. Die letztere Möglichkeit ist sogar zu bevorzugen, da hier die Grenze explizit angegeben und die Übersicht eher gewährleistet ist. Wollen Sie eine Schleife konstruieren, die bis zum Wert 100 zählt, so ist es empfehlenswert, genau diese Zahl auch in der Abfrage zu verwenden, z.B.

```
while(zaehler <= 100)
  ...
```

Beachten Sie bei diesen Vergleichsoperatoren, daß es bei "<=" und ">=" nicht in Ihr Ermessen gestellt ist, ob das Gleichheitszeichen vor oder hinter dem anderen Vergleichszeichen kommt. Das Zeichen "=" steht immer am Ende!

Ein praktisches Beispiel zeigt die Berechnung der Fakultät einer Zahl durch ständiges Multiplizieren. Die Fakultät hat in der Mathematik nichts mit einer Hochschule zu tun, sondern ist lediglich das Produkt aller ganzen Zahlen bis zu einem angegebenen Wert. So ist die Fakultät von 4:

$$4! = 1 * 2 * 3 * 4 = 24$$

```
main()
{
  int bis, i;
  float fakultaet;

  printf("Bitte geben Sie eine Zahl ein: ");
  scanf("%d", &bis);
  i = bis;
  fakultaet = 1;    /* Initialisieren */
  while(i >= 1)
    {
       fakultaet = fakultaet * i;
       i = i - 1;
    }
  printf("%d! = %f\n", bis, fakultaet);
}
```

An die Aztek-Benutzer: Denken Sie an "m.lib" beim Linken!

5.2 Die for-Schleife

Eine andere Schleife ist mit for zusammenzubasteln. In BASIC gibt es den gleichen Befehl, der dort folgendermaßen benutzt wird:

```
FOR I = 0 TO 100 STEP 2
  ...
NEXT I
```

Dies sieht in C so aus:

```
for(i = 0; i <= 100; i = i + 2)
  ...
```

Das NEXT, wie es in BASIC Verwendung findet, ist in C nicht nötig, da - wie bei allen Schleifen - lediglich der dem Schleifenkopf folgende Anweisungsblock ausgeführt wird. Innerhalb der Klammern tummeln sich aber einige interessante Anweisungen, die es genauer zu untersuchen gilt. Insgesamt finden sich

dort drei separate Anweisungen, die durch jeweils ein Semikolon voneinander getrennt sind. Hinter dem letzten Eintrag steht allerdings kein solches Trennzeichen, hier endet der Schleifenkopf durch die schließende runde Klammer. Der erste Eintrag "i = 0" dient in der Regel dazu, einer Variablen, die innerhalb der Schleife verändert wird, einen Startwert zuzuweisen. Die folgende Anweisung "i <= 100" stellt die Abbruchsbedingung dar. Solange sie erfüllt ist, wird die Schleife ausgeführt. Der letzte Teil des Schleifenkopfes dient dann zum Hoch- oder Herunterzählen der sogenannten Laufvariablen, die ja zuvor schon mit einem Startwert initialisiert wurde.

5.3 Die do-while-Schleife

Die letzte Schleifenart in C ist die do-while-Schleife. Sie kennen sich ja bereits mit der while-Schleife bestens aus, so daß diese angeblich neue Schleifenkonstruktion durch ein paar Bemerkungen abgehandelt werden kann. Vergleichen Sie bitte einmal folgende zwei Programmausschnitte:

1.)

```
while(i > 0)
  {
    i = i - 1;
    printf("I ist %d\n", i);
  }
```

2.)

```
do
  {
    i = i - 1;
    printf("I ist %d\n", i);
  } while(i > 0);
```

Zu Beginn der do-while-Schleife steht dort das "do", während das "while" mit der Abbruchsbedingung an den Schluß der Schleife gerückt ist. Dies führt zu einem kleinen, aber entscheidenen Unterschied im Programmablauf. Im ersten Beispiel überprüft das Programm, ob die Variable i noch größer als 0 ist und führt die Schleife nur dann aus, wenn diese Bedingung erfüllt ist. Im zweiten Beispiel erfolgt dieser Test erst nachdem die Schleife bereits einmal ausgeführt wurde. Enthält "i" z.B. den Wert 0, so wird die while-Schleife übersprungen, im Gegensatz zur do-while-Konstruktion, die auch in diesem Fall zumindest einmal durchlaufen wird.

Beachten Sie hier das Semikolon, das hinter dem while stehen muß. Es wird gerne vergessen, da bei der normalen while-Schleife auch kein Semikolon auftaucht.

Ich hoffe, Sie glauben mir das Gesagte auch ohne ausführliches Beispielprogramm. Wenn nicht, dann schreiben Sie doch selbst ein kleines Programm, das z.B.den Wert der benutzten Variablen ausgibt.

5.3.1 Schnitzeljagd

Ich habe für Sie aber auch noch eine andere Aufgabe auf Lager. Bislang konnten Sie kaum Fehler machen, es sei denn, Sie haben sich beim Abtippen vertan. Wenn Sie bein einem Programm plötzlich Fehlermeldungen erhalten, brauchen Sie nicht gleich in Panik zu geraten. Studieren Sie genau das, was der Compiler an dem Source auszusetzen hat. Ein wenig Fingerspitzengefühl und Intuition gehört natürlich auch dazu, um rafiniert getarnte Fehler zu entdecken. Dieses Gefühl wollen wir jetzt etwas schulen, indem Sie ein fremdes Programm analysieren. Hier sollen Sie selbst überlegen, wo Fehler versteckt sein können und an welcher Stelle Schwierigkeiten auftreten können. Dazu habe ich folgendes fehlerhaftes Listing zusammengestrickt, welches zu einer langen Liste von Fehlermeldungen führt. Sie sollten zuerst versuchen, die versteckten Fehler selbst zu finden, anschließend können Sie Ihren Compiler mit dieser Aufgabe betrauen. Die Auflösung finden Sie selbstverständlich auch in diesem Buch

und zwar direkt hinter dem Listing. Alle nötigen Informationen stehen also vor dem Listing. Das Programm soll alle Zahlen von 1 bis 100 addieren und die Summe auf dem Bildschirm inclusive der Zwischensummen anzeigen. Viel Erfolg bei der Fehlersuche!

```
main();
{
  printf("Ich addiere alle Zahlen von 1 bis 100/n");

  i = 1;
  do
    printf("Zwischensumme beim %d. Wert: %d/n", i, summe);
    summe = summe + i;
    i = i + 1;
  while(i < 100)
  printf("Summe aller Zahlen bis 100 ist %d/n", summe);
}
```

Na, haben Sie ein paar Fehler gefunden? War ja wohl auch nicht schwer, es ist ja fast alles falsch! Selbst wenn Sie keine Fehler gefunden haben, können Sie ohne Probleme der weiteren Materie folgen. Sie können dieses Kapitel auch überspringen, es zeigt Ihnen aber, wie leicht man Fehler machen oder übersehen kann. Und das ist auch wichtig, wenn Sie C an einem Wochenende lernen wollen.

Beginnen wir mal in der ersten Zeile, die ist nämlich auch schon fehlerhaft. Das Semikolon hinter "main" hat dort nichts verloren. Die nächste Zeile mit der geschweiften Klammer ist ausnahmsweise richtig. Der erste printf-Aufruf wird vom Compiler anstandslos geschluckt. Es ist kein syntaktischer Fehler enthalten, also eine für den Compiler unverständliche Anweisung. Die Zeile wäre sogar richtig, wenn wir wirklich am Ende der Zeile einen Schrägstrich und ein einsam dort stehendes "n" auf dem Bildschirm haben wollen. Wie Sie bestimmt erraten haben, sollte diese Kombination aber den Rechner dazu veranlassen, den nächsten Text in die folgende Zeile zu schreiben. Hierfür muß der Backslash "\" benutzt werden, und nicht der auf der Taste "7" plazierte Schrägstrich. Um es einheitlich zu halten, tritt dieser Fehler bei allen printf-Aufrufen auf.

Weiter geht's! Die Zuweisung "i = 1;" ist richtig, Applaus, Applaus! Bei der do-while-Schleife, die drei Befehle ausführen soll, fehlen die geschweiften Klammern um die drei Zeilen zu einem Anweisungsblock zusammenzufassen. Hinter dem while fehlt dann noch das abschließende Semikolon.

Innerhalb der Schleife sollen die Werte von "i" und "summe" ausgegeben werden. Außer dem Fehler beim Steuerzeichen "\n" ist hier alles in Ordnung. Anschließend wird der Wert von Summe um den Inhalt von "i" erhöht. Na, merken Sie was? Nicht? Was steht denn in der Variablen Summe? Tja, das wissen wir gar nicht, der Computer also auch nicht. Es wäre also nicht von Nachteil, würden wir die Variable mit 0 initialisieren. Olala, da fällt uns doch auf, daß wir die Variablen noch nicht einmal definiert haben, der Compiler weiß überhaupt nicht, was er sich unter "i" und "summe" vorzustellen hat. Wir müssen deshalb vor der ersten printf-Anweisung noch die Zeile

```
int summe, i;
```

einschieben. Vor oder hinter "i = 1;" muß noch irgendwo "summe = 0;" eingefügt werden. Nach diesen Änderungen freut sich zwar der Compiler, aber was das Programm anschließend auf dem Monitor anzeigt, kann nicht überzeugen. Es befinden sich noch zwei logische Fehler im Programm. Der erste tritt bei der Anzeige der Zwischensumme zutage. Der Wert von "summe" wird ausgegeben, bevor er überhaupt berechnet wurde. Die Zeile "summe = summe + i;" muß also vor der Zeile mit

```
printf("Zwischensumme beim %d. Wert: %d\n", i, summe);
```

plaziert werden. Schließlich bleibt noch die Abfrage in while übrig, die das Programm veranlaßt, nach der Zahl 99 das Summieren abzubrechen. Die Änderung ist einfach:

```
} while(i <= 100);
```

Das fehlerfreie Programm zeigt nun alle durchgeführten Korrekturen:

```
main()
{
  int summe, i;
  printf("Ich addiere alle Zahlen von 1 bis 100\n");

  summe = 0;
  i = 1;
  do
    {
       summe = summe + i;
       printf("Zwischensumme beim %d. Wert: %d\n", i, summe);
       i = i + 1;
    } while(i <= 100);
  printf("Summe aller Zahlen bis 100 ist %d\n", summe);
}
```

5.4 UND oder ODER

Bei der Benutzung der Schleifen geht es erst richtig rund. Leider können wir nur eine einzige Bedingung, die zum Abbruch der Schleife führt, angeben. Das ändert sich nun durch Einführung der Operatoren && und ||. Die "|"-Taste befindet sich auf der rechten Seite der Tastatur oberhalb der Return-Taste. Sie sehen nicht doppelt, es sind jedesmal zwei gleiche Zeichen. Das "&&" stellt das logische UND, das "||" das logische ODER dar. Warum die Operatoren unbedingt logisch sein müssen, erfahren Sie später, da es auch noch andere Und- und Oder-Operatoren in C gibt. Aus BASIC kennen wir die Befehle AND und OR, die äquivalent mit den Operatoren in C sind.

Verknüpfen wir einfach zwei Bedingungen durch UND:

```
while(i <= 10 && i >= 5)
  ...
```

Die Schleife wird nur ausgeführt, wenn

1.) i kleiner oder gleich 10 ist
2.) i größer oder gleich 5 ist

Ist eines der beiden Kriterien nicht erfüllt, z.B. bei i = 4, so ist die gesamte Bedingung falsch, also nicht erfüllt. Nur wenn beide Tests wahr sind, kann die Schleife ausgeführt werden.

Das ODER wird ebenfalls wie in der Umgangssprache benutzt. Wenn eine der angegebenen Bedingungen wahr ist, ist auch der gesamte Ausdruck wahr. Nehmen wir an, wir wollten eine Zeichenvariable auf bestimmte Inhalte testen. Da wir diese logischen Verknüpfungen auch bei anderen Bedingungsabfragen verwenden können, benutzen wir sie diesmal zusammen mit if:

```
if(operator == '+' || operator == '-' || operator == '*' ||
  operator == '/')
  printf("Der Operator ist gültig!\n");
```

Hier werden vier Abfragen getätigt, von denen nur eine zutreffen muß. Wenn mehrere Tests positiv sind, ist dies auch kein Problem, es genügt ja bereits eine wahre Bedingung. Daß wir obigen if-Befehl in zwei Zeilen schreiben können, dürfte für Sie nichts Neues sein. Schließlich wurde bereits zu Beginn deutlich, daß die Formatierung des C-Listings für den Compiler völlig egal ist.

Wir haben aber noch einen ganz tollen Operator auf Lager. Es ist der Negationsoperator "!", der bereits als Bestandteil von ungleich "!=" aufgetaucht ist. Mit diesem Zeichen lassen sich alle Aussagen und Überprüfungen in ihr Gegenteil umkehren. Wollen Sie wissen, ob ein Zeichen keinen gewünschten arithmetischen Operator enthält, so basteln wir aus obiger Aussage:

```
if( !( operator == '+' || operator == '-' || operator == '*' ||
  operator == '/'))
  printf("Kein gültiger Operator!\n");
```

Wie Sie sehen, wurden in diesem Fall alle Abfragen in einer runden Klammer zusammengefaßt. Wenn der Ausdruck innerhalb dieser Klammern nun wahr ist, also ein gültiges Zeichen vorliegt, so kommt der Negationsoperator zum Einsatz. Er verdreht den Sachverhalt einfach. Aus der wahren Aussage, die wir gerade bekommen haben, macht er eine falsche, so daß der printf-Befehl nicht ausgeführt wird. Ähnlich sieht es aus, wenn innerhalb der Klammern eine falsche Aussage zustande kommt, also keines der Zeichen "+-*/" in der Variablen gespeichert ist. In diesem Fall macht der "!"-Operator daraus eine wahre Aussage. Das ist die gleiche Prozedur, als wenn Sie in irgendeinen Satz ein "nicht" einbauen. Sie können in der Umgangssprache sogar mehrere Verneinungen in einem Satz benutzen, ob Sie allerdings dann noch jemand versteht, ist eine andere Frage.

6. Strings durchleuchtet

Wenden wir uns nun noch ein wenig den Zeichenketten zu. Nach der Eingabe eines Strings über Tastatur haben wir ihn lediglich wieder ausgegeben. Was sollten wir denn sonst damit machen?

Eine Zeichenkette wird ja, falls Sie ein gutes Gedächtnis besitzen, mit

```
char name[felder_anzahl];
```

definiert. Daran können wir schon erkennen, daß es sich bei Strings nicht um eine untrennbare Einheit wie z.B. um einen Integerwert handelt, sondern daß mehrere kleinere Teile (char) unter einem einzigen Begriff abgelegt werden. In der Definition der Variablen geben wir dem Compiler zu erkennen, wie viele Bestandteile unser String maximal erhalten soll. Wenn wir ein einzelnes Zeichen daraus bearbeiten wollen, genügt ja nicht der Name allein. Zusätzlich braucht man die Information, an welcher Stelle oder Position es zu finden ist. Diese Angabe, also die Nummer des Zeichens, wird in eckige Klammern gefaßt, genau so, wie wir es bereits bei der Definition vorgenommen haben.

Nehmen wir uns das erste Zeichen eines Strings vor. Es steht damit an erster Stelle und besetzt die Position mit der Nummer 0. Der Computer beginnt beim Zählen nämlich immer mit 0, wodurch sich auch alle folgenden Stellen um eins verschieben. Das zweite Zeichen kann mit dem Index, so nennt man die Positionsangabe eines Feldes, mit dem Wert 2 erreicht werden. An jeder Position findet ein Zeichen Platz. Alle Buchstaben sind fein säuberlich hintereinander in einer mehr oder weniger langen Kette angeordnet.

6.1 Rückwärts geht's weiter

Um etwas Praxis in die Materie zu bringen, schreiben wir ein überaus sinnvolles Programm, das den eingegebenen Text rückwärts auf den Bildschirm ausgibt. Bevor wir damit anfangen können, müssen wir noch eine sehr wichtige Voraussetzung beachten. Ein String kann ja beliebig lang sein, am Ende ist deshalb immer ein Eintrag mit dem Wert 0. Auf den müssen wir stets achten!

Beginnen wir, uns ein paar Gedanken über unser Programm zu machen. Wenn man den Text, den man mittels scanf eingegeben hat, rückwärts ausgeben soll, müssen wir logischerweise beim letzten Zeichen beginnen. Es stellt sich die Aufgabe, die Nummer des letzten Zeichens zu ermitteln. Nehmen wir dazu eine kleine for-Schleife, die von vorne nach hinten alle Zeichen prüft, ob nicht vielleicht der Wert 0 darinsteht. Wie wäre es damit?

```
...
char eingabe[81];
int index;

for(index = 0; eingabe[index] != 0; index = index + 1)
  ;
...
```

Richtig schnuckelig, diese kleine Schleife! Dabei wird sie noch gar nicht voll ausgeschöpft. Der Schleifenrumpf, die Befehle, die bei jedem Schleifendurchlauf ausgeführt werden sollen, ist nämlich leer. Nach der for-Schleife steht einsam und verlassen ein Semikolon. Da nach jeder Schleife eine Anweisung oder gar ein Anweisungsblock folgt, stellt hier ein Semikolon einen Befehl dar, der nichts tut. Dies ist die sogenannte leere Anweisung. Alle nötigen Operationen werden innerhalb der runden Klammern erledigt. Zuerst wird der Index auf Null gesetzt. Anschließend erfolgt der Test, ob noch ein Zeichen ungleich dem Wert 0 enthalten ist, und dann, wenn diese Bedingung erfüllt ist, wird der Index um 1 erhöht. Beim letzten Element dieser Zeichenkette, das 0 beinhaltet, bricht die Schleife ab, und wir haben die Nummer dieses Eintrages. Aufgabe Nummer 1 gelöst!

Jetzt pirschen wir uns von hinten an den String heran. Das letzte Zeichen (außer der 0), liegt eine Position davor. "index" muß vor der ersten Verwendung erst wieder um eins verringert werden. Nun zählen wir den Index Schritt für Schritt auf 0 herunter und geben jedesmal ein Zeichen aus.

```
do
  {
    index = index - 1;
    printf("%c", eingabe[index]);
  } while(index > 0);
```

Auch das war kein Problem! Das übliche Drumherum, wie die Eingabe entgegennehmen und vielleicht noch einen Text ausgeben, ist auch Routinearbeit. Alles zusammen steht nun hier:

```
main()
{
  char eingabe[81];
  int index;

  printf("Jetzt dürfen Sie einen Text eintippen!\n");
  scanf("%s", eingabe); /* Strings benötigen kein & */

  for(index = 0; eingabe[index] != 0; index = index + 1)
    ;  /* Endemarkierung suchen! */

  printf("Ihre Eingabe >%s< hat %d Zeichen\n", eingabe, index);

  do
    {
       index = index - 1;
       printf("%c", eingabe[index]);
    } while(index > 0);

  printf("\n\n");  /* Leerzeile bevor das Prompt kommt */
}
```

Ganz ohne es zu merken haben wir bei unserer for-Schleife die Länge des Eingabestrings berechnet. Vielleicht kennen Sie als (ehemaliger) BASIC-Programmierer die LEN-Funktion. In C läßt sich so eine Routine mit Leichtigkeit selbst schreiben.

C durch und durch

7. Rechnen in C

Im Einstiegsteil haben wir bereits ausführlich von der Rechenkunst unseres AMIGAs Gebrauch gemacht. Addition (+), Subtraktion (-), Multiplikation (*) und die Division (/) sind uns vertraut. Ein weitere Rechenoperation kann mit dem Modulo-Operator "%" durchgeführt werden. Er errechnet den Rest einer Division. Die Zuweisung des Ergebnisses erfolgt durch das Gleichheitszeichen "=", wobei die Variable links vom Zuweisungsoperator stehen muß. Die allgemeine Form lautet:

Variable = operand1 # operand2
(# stellt den Operator dar)

Zuerst wird der gesamte Ausdruck rechts vom Gleichheitszeichen berechnet und anschließend der Variablen links davon zugewiesen. Daher sind auch Anweisungen wie

```
variable = variable + 1;
```

möglich, die zwar mathematisch unerfüllbar sind, für den Rechner aber keine Probleme bieten. Der Variableninhalt wird geholt, eins hinzuaddiert und das Resultat wieder in derselben Variablen abgelegt. Es sind aber auch Kombinationen von arithmetischen Operationen möglich, wie die folgenden gültigen Beispiele zeigen:

```
zahl = 3 * 32;
zahl = 2 + 6 * 7;
zahl = 5 * (180 / 3 + 9) * (5 - 2);
zahl = zahl - 1;
zahl = zahl % 2;
```

Die Division und Multiplikation haben nach der Punkt-vor-Strich-Regel Vorrang vor Addition und Subtraktion. Diese Regel wird auch vom C-Compiler beachtet, so daß in "2 + 6 * 7" keine Klammern gesetzt werden müssen, um die richtige Bearbeitung des Terms zu gewährleisten. Der Modulo-Operator, der den Rest einer Division liefert, hat damit die gleiche Priorität wie "/", also Vorrang vor Addition und Subtraktion. Beispiel:

5 % 3 = 2, da 5 / 3 = 1 Rest 2 ist.

Bei so einfachen Programmteilen, die lediglich ein Ergebnis berechnen und dieses anschließend mittels einer Funktion verwerten, braucht nicht unbedingt eine Variable verwendet zu werden. Sehen wir uns dazu einmal folgendes einfaches Beispielprogramm an:

```
main()
  {
    int zahl;
    zahl = 3 * 12;
    printf("Ergebnis: %d\n", zahl);
  }
```

Die Variable ist im oben gezeigten Programm zu umgehen, da der printf-Funktion durch "%d" mitgeteilt wird, daß sie einen Integerwert zu erwarten hat. Wir können deshalb den Term "3 * 12" direkt als Parameter der Funktion übergeben. Die Berechnung des Ergebnisses erfolgt nämlich, bevor der Wert übergeben wird, so daß wir dafür keine Variable benötigen. Das folgende Programm ist dadurch kürzer und schneller:

```
main()
  {
    printf("Ergebnis: %d\n", 3 * 12);
  }
```

Weil eine Integerzahl nur ganze Zahlen aufnehmen kann, erhalten wir nach der Zuweisung von "zahl = 3 / 2" in der Variablen "zahl" den Wert 1. Das korrekte Ergebnis wäre 1.5, da dies aber keine ganze Zahl darstellt, wird auf die nächste ganze Zahl abgerundet, nämlich 1. Aber Achtung: Die Zahl -2.25 wird aber auf -2 gerundet und nicht, wie vielleicht vermutet auf -3, da ja -2 größer als -3 ist. Beim Lattice-C wird zur 0 hin gerundet, das kann je nach Compiler auch anders sein. Hier hilft nur ein Probelauf mit beispielsweise "-5 / 2", bei dem entweder -2 (zur Null hin) oder -3 (abgerundet) als Ergebnis erscheint.

8. Variablen

Variablen, die bereits zum Abspeichern diverser mathematischer Ergebnisse benutzt wurden, unterliegen bestimmten Gesetzen. So muß ihnen ein bestimmter Datentyp (z.B. int) und ein eindeutiger Name zugewiesen werden.

8.1 Variablennamen

Der Name, den Sie der Variablen mit auf den Weg geben, muß einigen Regeln gerecht werden. Diese sind in einer Stichwortliste zusammengetragen:

1. Das erste Zeichen ist ein Buchstabe (das Unterstreichungszeichen '_' zählt als ein solcher).
2. Es dürfen keine Sonderzeichen oder Umlaute darin vorkommen.
3. Nach dem ersten Zeichen können zusätzlich zu den Buchstaben auch Ziffern verwendet werden.
4. Die Namen einer Variablen können beliebig lang sein, es werden aber nur die ersten 8 Zeichen unterschieden. (Der Lattice-C-Compiler läßt sogar bis zu 30 signifikante Zeichen zu.)
5. Es dürfen keine Schlüsselworte von C als Variablenname verwendet werden.
6. Groß- und Kleinschrift wird unterschieden

Um die Regeln zu verdeutlichen einige Beispiele von Variablennamen, die korrekt oder fehlerhaft gebildet wurden. Können Sie die falsch zusammengestellten Namen finden?

a) Zahl_1
b) 2_pi
c) erste-var
d) Buch_nr_1
e) Buch_nr_2
f) int
g) _flag
h) int_wert

i) zahl_1
j) geheimes_Passwort

Korrekt sind die Variablennamen a), d), e), g), h), i) und j). Dabei ist zu beachten, daß a) und i) verschiedene Variablen sind, da Großschreibung von Kleinschreibung unterschieden wird. d) und e) wiederum beziehen sich auf die gleiche Variable, da sich die Namen nicht in den ersten 8 Stellen unterscheiden. Die Namen dürfen ruhig länger sein, ohne daß daraus eine Fehlermeldung resultiert: Beispiele d), e) und j). Bei h) wurde zwar ein Schlüsselwort von C im Variablennamen verwendet, dies ist aber ohne weiteres zulässig.

Fehlerhaft sind (logischerweise) die restlichen 3 Beispiele b), c) und f). Die Variable bei b) beginnt mit einer Ziffer, das wird ihr zum Verhängnis. c) enthält ein Sonderzeichen, nämlich '-' und schließlich f), das als Variablennamen ein reserviertes C-Schlüsselwort darstellt. Die Liste dieser speziellen C-Befehle ist nicht besonders groß:

auto	enum	short
break	extern	sizeof
case	float	static
char	for	struct
continue	goto	switch
default	if	typedef
do	int	union
double	long	unsigned
else	register	void
entry	return	while

Nachdem nun klar sein dürfte, wie man Variablennamen bildet, wollen wir uns nun damit befassen, welche Datentypen es gibt, also was in eine Variable hineingepackt werden kann.

8.2 Datentypen

Wir haben bei den ersten Versuchen bereits drei Datentypen kennengelernt: int, float und char (Strings).

Fassen wir noch einmal zusammen: int steht für Integerwerte, das sind alle ganzen Zahlen zwischen -32768 und 32767. Für Gleitkommazahlen gibt es den Datentypen float. Eine float-Variable eignet sich zum Speichern extrem großer oder kleiner Zahlen, außerdem für Zahlen mit Nachkommastellen. Bei dieser Art von Variablen können die Werte auch in wissenschaftlicher Notation dargestellt werden. Sehr kleine Zahlen wie 0.00000015 können auch durch 15E-7 dargestellt werden. Die Schreibweise mit dem "E" ist eine Abkürzung für:

$$15 * 10^{-7}$$

Dadurch lassen sich auch sehr große Zahlen, die vielleicht mehr als 15 Stellen besitzen, knapp und präzise darstellen. Das "e", das den Exponent (hier 7) von der Mantisse (in diesem Fall die 15) trennt, kann groß oder klein geschrieben werden. In beiden Fällen ist eine einwandfreie Übersetzung durch den Compiler gewährleistet.

Aber auch eine solche Variable hat ihre Grenzen bei zulässigen Werten: Die dem Betrag nach größtmögliche Zahl liegt bei 10^{38}, ab 10^{-38} wird alles, was dem Betrag nach kleiner ist, als 0 behandelt. Der Wert 10^{-40}, ausgeschrieben eine Zahl, die zuerst 39 Nullen hinter dem Komma aufweist, bevor die eins kommt, wird vom Rechner einfach als 0 angesehen. float-Variablen sind in der Regel auf ca. 7 Stellen genau. Bei unserem ersten Rechenprogramm dürfte Ihnen das aufgefallen sein, wenn Sie ein bißchen damit herumgespielt haben. Vielleicht versuchen Sie es einmal mit der Zahl 16.8, die vom Programm automatisch in 16.799999 umgerechnet wird.

Wer eine höhere Genauigkeit benötigt, kann double-Variablen verwenden, die normalerweise doppeltgenau wie float-Variablen sind (ca. 11 - 14 Stellen). Der dazugehörige Datentyp ist "double". Sie benötigen dafür allerdings auch mehr Speicherplatz.

Bei der Definition einer Variablen, also der Festlegung, welchen Datentyp sie darstellt, benutzt man für float-Variablen das C-Schlüsselwort "float", für double-Variablen "double". Aber auch diese Variablen sind nicht so genau wie beispielsweise Integerwerte. Auch ein sehr geringer Rundungsfehler kann schlimme Auswirkungen haben, wenn er z.B. bei vielen Rechnungen benutzt wird. Das Ergebnis wird mit jeder weiteren arithmetischen Operation ungenauer. Deshalb sollten Sie sich hüten, bei einer Abfrage, z.B. bei if, mit einem bestimmten festgelegten Wert zu vergleichen.

```
if(wert == 1.0) /* So nicht! */
  ...
```

Besser ist es, gleich auf größer oder kleiner abzufragen, damit der überprüfte Grenzwert nicht durch einen Rundungsfehler übersprungen werden kann. Die Folge wäre, daß eine so ausgestattete Schleife nie wieder verlassen würde.

Bei der Benutzung von Fließkommazahlen gibt es noch eine wichtige Besonderheit: Um festzulegen, daß es sich bei einer Konstanten ohne Nachkommastellen (z.B. 2) um eine Fließkommazahl handelt, muß an die Zahl noch eine Stelle hinter dem Komma angehängt werden, also 2.0. Bei dem folgenden Aufruf von printf tritt dieser Fehler hervor:

```
printf("Ergebnis von 2 / 3 = %f.\n", 2/3);
```

2/3 wird als Integerwert berechnet, sodaß als Ergebnis 0 übergeben wird. Die Funktion erwartet aber einen Floatwert, was durch "%f" mitgeteilt wird. Richtig wäre diese Zeile in der Form:

```
printf("Ergebnis von 2 / 3 = %f.\n", 2.0/3.0);
```

Aber Vorsicht: Sie erhalten in der Regel nicht nur ein völlig falsches Ergebnis, sondern können meistens auch einen Rechnerabsturz live miterleben. Sollten Sie das Bedürfnis verspüren, wirklich auszuprobieren, wo nun der Unterschied der beiden Formulierungen liegt, dann retten Sie vorher alle wichti-

gen Daten (z.B. die empfindliche RAM-Disk)! Eine Meldung vom Oberguru ist dann die gerechte Quittung für derartige Programme.

Aus den Basistypen int und float lassen sich alle weiteren anderen Datentypen ableiten. So stellt beispielsweise der Typ "char", der ein Zeichen aufnehmen kann, eigentlich eine Variable für ganze Zahlen zwischen -128 und 127 dar. Wir haben dadurch einen kleineren Verwandten von "in" vor uns. Eine Nummer größer als "int" ist der Datentyp "long".

long ist ebenfalls eine Integerzahl, die Zahlen zwischen -2147483648 und 2147483647, also auch sehr große Zahlen aufnehmen kann. Sollen Konstanten als solche long-Werte definiert werden, so wird anstelle des ".0" bei float-Werten, hinter die Zahl ein "I" oder "L" gestellt, beispielsweise

```
1L
```

Desweiteren können die C-Schlüsselworte unsigned und short, genauso wie long als Adjektive zu den Basistypen verwendet werden. Dadurch wird ein Integerwert genauer spezifiziert. Dabei sind aber die Kombinationen von unsigned und short für float-Werte unzulässig, da unsigned eine Integerzahl definiert, die kein Vorzeichen besitzt, und short eine kleine ganze Zahl aufnehmen soll.

Von den vielen verschiedenen Möglichkeiten sind nur wenige wirklich sinnvoll. Fehlt die Angabe von int oder float, wird das Vorhandensein von int vorausgesetzt, so daß folgende Kombinationen gültig sind:

unsigned = unsigned int
short = short int = (Compiler-abhängig) char
long = long int
unsigned long int
long float = double

Vorteile bringt die Festlegung auf unsigned, also die Beschränkung auf positive Werte, durch den größeren Wertebereich für positive Zahlen, der sich verdoppelt. unsigned int kann z.B. Zahlen zwischen 0 und 65535 aufnehmen, während int normalerweise nur positive Wert bis 32767 zuläßt. Außerdem läßt sich mit diesen Zahlen auch noch allerhand anstellen, das mit anderen Typen nicht durchzuführen ist. Dazu später mehr.

Die Wertzuweisung an char-Variablen erfolgt in gleicher Weise, wie es auch mit int-Werten gehandhabt wird, durch

```
char zeichen;
zeichen = 65;
```

Bei solchen Variablen ist es aber sinnvoll, das Zeichen, welches darin gespeichert werden soll direkt zuzuweisen. Dies erfolgt durch folgenden Ausdruck, der zum selben Resultat wie die obige Zeile führt.

```
zeichen = 'a';
```

Die Zusammenstellungen von Integerwerten mit den Attributen short, long und unsigned liefern von Compiler zu Compiler unterschiedliche Resultate, daher kann kein allgemeingültiger Wertebereich oder Speicherplatzbedarf angegeben werden. Die einzige Relation, die von jedem Compiler eingehalten wird, die diese Datentypen implementiert haben, ist die folgende:

char <= short <= int <= long

Für die beiden Compiler gilt die folgende Belegung:

	Lattice	Aztek
char	1	1
short	2	2
int	4	2
long	4	4

8.3 Typumwandlungen

Durch die Verwendung von verschiedenen Datentypen werden in Berechnungen zahlreiche Typumwandlungen erforderlich. Diese werden nach folgenden Regeln vollzogen:

1. char und short werden immer in int und float in double umgewandelt.
2. Sollte nach dieser Umwandlung einer der Operatoren den Typ double haben, so werden der zweite Operand und das Ergebnis ebenfalls in double umgewandelt.
3. Wenn ein Datentyp jetzt long ist, werden alle beteiligten Werte auch in long transformiert.
4. Sollte sich unter den Operanden noch ein unsigned-Wert befinden, erfolgt die Umwandlung aller Werte in unsigned.

8.4 Die cast-Anweisung

Sie können Konstanten und Variablen sowie Funktionswerte aber auch "gewaltsam" in einen gewünschten Datentyp umwandeln. Der umzuwandelnde Parameter wird in runde Klammern gesetzt und erhält davor, ebenfalls in runden Klammern, den Datentyp gestellt, den cast-Operator. Die Klammern für den Parameter sind nicht vorgeschrieben, aber wegen der hohen Priorität der Typumwandlung empfehlenswert. Z.B.

```
long zahl;
zahl = 123 / (long)('a' / 1.5);
```

Allgemein ausgedrückt lautet der Ausdruck:

(typ) Parameter

oder besser (sicher ist sicher)

(typ)(Parameter)

Für "typ" können alle erdenklichen Datentypen eingesetzt werden.

9. printf und scanf

Die wohl leistungsfähigste Ausgabefunktion in C ist die printf-Routine. In dem Steuerstring, dem ersten Parameter der printf-Funktion, können folgende Steuerzeichen mit unterschiedlichen Auswirkungen auftauchen:

9.1 Steuerzeichen

- \t die nächste Ausgabe erscheint am nächsten Tabulatorstop (alle 8 Stellen)
- \b bewegt die aktuelle Schreibposition um eine Stelle nach links (backspace)
- \r Wagenrücklauf, an die erste Position der aktuellen Zeile (carriage return)
- \n Wagenrücklauf und neue Zeile (newline)
- \f Seitenvorschub (form feed)
- \\ Das Zeichen '\' selbst (backslash)
- \" Anführungszeichen innerhalb der Zeichenkette
- \' Einfaches Anführungszeichen
- \nnn Beliebiges Zeichen mit dem Oktalwert "nnn"

Wichtig: Ein Steuerzeichen wird zwar mit 2 Zeichen im Text untergebracht, stellt aber nur ein einziges Zeichen dar. Bei Speicherplatzberechnungen sollten Sie dies beachten!

Das folgende Programm benutzt das Steuerzeichen '\t' für Tabulatorstops:

```
main()
  {
    printf("Ein\tBeispiel,\two\tTabulatoren\tden");
    printf("\tText\tplazieren!\n");
  }
```

Nun stellt sich vielleicht die Aufgabe, den folgenden Text auf den Bildschirm zu transportieren:

```
Benutzen Sie die Steuerzeichen : "\n", "\t"!
```

Einfach den Text in Anführungszeichen zu setzen, scheitert daran, daß im Text selbst diese Zeichen vorkommen. Hierfür werden, wie könnte es anders sein, Steuerzeichen eingesetzt. Das Hochkomma erhält man durch \", den Backslash durch \\. Der printf-Aufruf sieht also so aus:

```
printf("Benutzen Sie die Steuerzeichen : \"\\n\", \"\\t\"!");
```

Das sieht zwar etwas unübersichtlich aus, es funktioniert aber tadellos. Versuchen Sie doch jetzt einmal herauszubekommen, was das folgende Programm auf den Monitor zaubert, bevor Sie es starten!

```
main()
  {
    printf("Kleines ");
    printf("\"T e s t p r o g r a m m");
    printf("\"\n\nWo\rsteht hier\njetzt wen\b\bas?\n");
    printf("\tAlles klar\r\tSonnen\n");
  }
```

Falls Sie sich nicht die Mühe machen wollen, es auszuprobieren, hier die Lösung zum Vergleichen:

```
Kleines "T e s t p r o g r a m m"

steht hier
jetzt was?
        Sonnenklar
```

9.2 Formatanweisungen bei printf und scanf

Außer reinem Text und Steuerzeichen können Sie im Steuerstring auch Formatanweisungen unterbringen. Zu jeder Formatanweisung folgen dann im Anschluß an den String die entsprechenden Variablen, die durch Komma getrennt an den String angehängt werden. Die Formatelemente beginnen stets mit einem Prozentzeichen und können sowohl in der printf-, als auch in der scanf-Funktion verwendet werden.

Ein Unterschied zum Formatstring der printf-Funktion ist aber wichtig: Die scanf-Funktion dient zum Einlesen der Daten, deshalb darf außer Formatanweisungen und Steuerzeichen wie Zeilentrenner '\n', Tabulatorzeichen '\t' und Leerzeichen, die allesamt ignoriert werden, nichts anderes in dem String enthalten sein.

Hier nun eine Tabelle mit allen Formatanweisungen für printf- und scanf-Funktion:

Formatelement	Datentyp
%c	char (ein Zeichen)
%d	Integerwert
%s	String (Zeichenkette)
%f	float- und double-Zahl (bei printf Ausgabe im Format: [-]xxx.xxxxxx)
%o	Integerwert als Oktalzahl (Basis 8)
%x	Integerwert als Hexzahl (Basis 16)
%u	(unsigned) Integerwert ohne Vorzeichen (nur printf)
%e	float- oder double-Wert, der im wissenschaftlichen Format: [-]x.xxxxxxE[+-]xx ausgegeben wird (printf) bei scanf wie "%f"
%g	gibt die kürzere Form von "%e" und "%f" aus (nur printf)
%h	short (nur für scanf)
%%	stellt das Zeichen '%' selbst dar (nur printf)

Zusätze

Vor die Elemente von Integers 'd', 'u', 'o' und 'x' kann der Buchstabe 'l' gesetzt werden, um anzuzeigen, daß nicht Integer, sondern long-Werte benutzt werden, die ja nur Integer mit doppelter Länge darstellen. Bei den Elementen für float-Zahlen 'e', 'f' und 'g' bewirkt ein vorangestelltes 'l', daß double-Werte erwartet werden.

Die Breite der Ein- oder Ausgabe eines Feldes kann ebenfalls mit einem Formatelement angegeben werden. Nach dem Prozentzeichen kann die Größe des einzelnen Feldes bestimmt werden. Ist das erste Zeichen dieser Zahl ein Minuszeichen, so wird der Text nach links ausgerichtet. Die restlichen Stellen des Feldes werden mit Leerzeichen aufgefüllt.

Ohne Angabe einer Feldbreite ist bei "%f" die Standardeinstellung der printf-Funktion "%.6f" eingeschaltet. Die Ausgabe erfolgt also stets mit 6 Nachkommastellen und beliebiger Feldbreite.

printf und Element für Floatwerte:

%<Min.Zeichen>.<Nachkomma>F

<Min.Zeichen> gibt die minimale Breite des Ausgabefeldes, <Nachkomma> die maximale Anzahl von Nachkommastellen an. F schließlich ist eines der Zeichen 'e', 'f', 'g' für das Format. Durch Angabe von 0 Nachkommastellen werden sämtliche Stellen hinter dem Komma abgeschnitten (z.B. mit "%.0f").

Beispiel:

```
printf("Zahl %5.2lf\n", 12.345);
```

erzeugt die Ausgabe:

```
Zahl 12.35
```

In diesem Fall wird eine double-Zahl ("lf") erwartet, die mindestens 5 Zeichen breit sein, aber nur maximal 2 Nachkommastellen haben soll. Da im obigen Fall auf die 2. Stelle hinter dem Komma gerundet werden muß, erscheint an letzter Stelle die Ziffer 5. Wenn weniger Nachkommastellen als angegeben vorhanden sind, werden entsprechend Nullen angehängt.

printf und Element für Integer:

%<Min>F

<Min> gibt die minimale Breite des Ausgabefeldes an, F stellt eines der folgenden Zeichen dar: 'd', 'u', 'o' oder 'x'. Beispiel:

```
printf(">%4d<", 12);
```

Ausgabe:

```
> 12<
```

printf und "%s":

%<Min>.<real>s

<Min> zeigt wieder die minimale Breite des Ausgabefeldes an, während <real> die wirkliche Anzahl auszugebender Zeichen beschreibt. Einige der vielfältigen Kombinationsmöglichkeiten sind in folgenden Beispielen anhand des String "Probetext" gezeigt:

Formatelement	Ausgabe
>%6s<	>Probetext<
>%-6s<	>Probetext<
>%12s<	> Probetext<
>%-12s<	>Probetext <
>%12.6s<	> Probet<
>%-12.6s<	>Probet <
>%.6s<	>Probet<

Bei der scanf-Funktion ist die Sache etwas einfacher. Dort existiert nur eine Zahl, die die maximal mögliche Eingabelänge festlegt. Sobald ein Zeichen nicht mehr in das Format eines bestimmten Datentypes paßt oder ein Steuerzeichen oder Leerzeichen auftaucht, ist die Eingabe für das aktuelle Feld beendet. Konkret bedeutet das bei der Eingabe einer Zahl, daß nur die Zeichen für eine Integerzahl benutzt werden, die eine Ziffer

darstellen. Tauchen andere Zeichen auf, ist die Eingabe für die Integerzahl abgeschlossen. Zusätzlich kann noch '*' verwendet werden, das vor dem Formatelement für den Datentyp gesetzt wird und die Zuweisung unterdrückt. Das entsprechende Feld wird in diesem Fall einfach übersprungen.

```
int i;
float f;
char string[50];
scanf("%3d %f %*d %s", &i, &f, string);
```

Eingabe: `1234567.89 12345alles klar?`

Wertzuweisungen:

i enthält 123, da das Feld 3 Stellen haben soll und keine anderen Zeichen als Ziffern auftauchen. f beinhaltet den Wert 4567.89, weil das Leerzeichen nach "9" das weitere Einlesen verhindert, genauso, wie es nach dem Speichern von "alles" in string[] abbricht. Die Ziffernfolge "12345", die normalerweise einem Integerwert zugewiesen worden wäre, wurde wegen des Sterns übersprungen. In der Praxis bedeutet dies, daß mit der scanf-Funktion keine Leerzeichen eingelesen werden können, und sie somit für Eingaben von String nicht gerade ideal ist.

Falls Sie nicht alles behalten haben, trösten Sie sich. Wir werden im weiteren Verlauf einige dieser Formatanweisungen intensiv benutzen, so daß sie bei Ihnen in Fleisch und Blut übergehen werden. Beginnen wir mit den Zeichen "o" und "x", die eine Zahl oktal oder hexadezimal ausgeben oder einlesen. Da stellt sich die Frage, was ist hexadezimal?

9.3 Oktal und Hexadezimal

Um das zu klären, muß etwas weiter ausgeholt werden. Beginnen wir mit dem vertrauten Zahlensystem, dem Dezimalsystem. Nehmen wir uns einmal eine Dezimalzahl vor und zerlegen sie in die Bestandteile:

$$
\begin{aligned}
&5279 \\
&= 5000 && + 200 && + 70 && + 9 \\
&= 5 * 1000 && + 2 * 100 && + 7 * 10 && + 9 * 1 \\
&= 5 * 10^3 && + 2 * 10^2 && + 7 * 10^1 && + 9 * 10^0
\end{aligned}
$$

Jetzt kann man erkennen, warum das Wort "dezimal" = 10 in unserem gebräuchlichen Zahlensystem seine Berechtigung hat. Jede Stelle einer Zahl hat einen bestimmten Wert, es gibt Einer, Zehner, Hunderter usw. Der Wert dieser Position wird mit der dort befindlichen Ziffer multipliziert, z.B. 7 * 10. Die Faktoren 1, 10, 100, 1000 lassen sich aber wiederum alle auf die Basis 10 zurückführen. Der Exponent der Basis 10 hängt von der Position der Ziffer in der Zahl ab, die erste Position entspricht Exponent 0, die zweite 1, die dritte 2 und so weiter und so fort.

Wie Sie wissen, können Sie alle Ziffern von 0 bis 9 verwenden, es stehen Ihnen damit 10 verschiedene Ziffern zur Verfügung. Hierdurch erklärt sich also die Basis 10.

Wenn wir jetzt anstelle von 10 verschiedenen Ziffern nur noch acht (0 - 7) verwenden, so beträgt die Basis bei unseren Berechnungen 8. Man befindet sich im Oktalsystem. Berechnen wir nun den Wert der folgenden Oktalzahl. Um die Zahlen aus verschiedenen Zahlensystemen unterscheiden zu können, stellt man die Basis in der Regel als Index oder in Klammern gefaßt hinter die Zahl.

6204_8

$= 6 * 8^3 \quad + 2 * 8^2 \quad + 0 * 8^1 \quad + 4 * 8^0$

$= 6 * 512 \quad + 2 * 64 \quad + 0 * 8 \quad + 4 * 1$

$= 3072 \quad + 128 \quad + 0 \quad + 4$

$= 3204_{10}$

So, nun kommen wir wieder auf das Formatzeichen zurück. Wenn Sie wollen, daß eine Variable in oktaler Schreibweise auf dem Bildschirm erscheint, so verwenden Sie "%o".

```
printf("3204 dez = %o okt\n", 3204);
```

Diese printf-Zeile nimmt uns die gerade per Hand durchgeführte Umrechnung ab.

Benutzen wir das Formatzeichen "%x", so bewegen wir uns im Hexadezimalsystem. Hier werden alle Berechnungen zur Basis 16 getätigt. Das bringt allerdings eine kleine Schwierigkeit. Aus unserem Dezimalsystem stehen 10 Ziffern (0 - 9) zur Verfügung, benötigt werden aber 16. Wo sollen wir die fehlenden 6 Ziffern herholen? Da greift man einfach in die Kiste mit dem Alphabet und holt sich die ersten 6 Buchstaben als Verstärkung heraus. Der Buchstabe A stellt nun die Ziffer 10 dar. Die Betonung liegt auf Ziffer! B hat dann den Wert 11, C 12, D 13, E 14 und F als höchstwertige Ziffer 15. Führen wir die schon erprobte Prozedur jetzt an einer hexadezimalen Zahl aus:

$5DA9_{16}$

$= 5 * 16^3 \quad + 13 * 16^2 \quad + 10 * 16^1 \quad + 9 * 16^0$

$= 5 * 4096 \quad + 13 * 256 \quad + 10 * 16 \quad + 9 * 1$

$= 20480 \quad + 3328 \quad + 160 \quad + 9$

$= 23977_{10}$

Übrigens können Sie hexadezimale und oktale Zahlen in C genauso verwenden, wie wir es mit Deziamlzahlen getan haben. Bei Hex-Zahlen ist durch die ersten zwei Zeichen "0x" oder "0X"

angezeigt, daß die Basis 16 zu Grunde liegt. Für das Oktalsystem benötigen Sie nur eine führende Null. Einige Beispiele:

```
0X5DA9
0xFFFF
0612
0x5da9
0x123
0815
0X5da9
06543
```

Eine der obigen Kombinationen ist falsch, bzw. macht nicht das, was sie eigentlich soll. Entdeckt? Schauen Sie sich die Zahlen noch einmal genau an. Unter der unscheinbaren Zahlenkombination "0815" hat sich der Fehler versteckt. Durch die führende Null soll dies ja eine Oktalzahl darstellen. In dem Achtersystem existiert aber keine Ziffer mit dem Wert "8". O.K.?

Es hängt natürlich auch etwas vom Compiler ab, welche Reaktion Ihnen entgegenschlägt. Es kann sein, daß der Compiler mitteilt, daß dort eine falsche Ziffer auftaucht, oder er sie als Dezimalzahl übernimmt oder (und das macht Lattice) die Zahl "mit Gewalt" vom Oktalsystem ins Dezimalsystem umrechnet. Dann kommt leider etwas völlig anderes heraus, nämlich 525_{10} gleich 1015_8.

Wenn Sie bei jeder Zahl, die Sie umrechnen wollen, die oben aufgezeigte umfangreiche Rechnung durchführen wollen, wird Ihnen bald die Lust vergehen. Wozu hat man denn für solche gleichartigen Aufgaben eigentlich einen Computer? Das wäre doch eine Aufgabe für ein praktisches C-Programm mit etwas Nährwert.

Wenn wir aber ein Programm schreiben, das Zahlen von verschiedenen Basen in unser gebräuchliches Dezimalsystem umrechnet, dann müssen wir die Vorgehensweise etwas ändern. Nichts ist einfacher als eine Schleife zu konstruieren, das kann eine Menge Tipparbeit ersparen. Die Überlegung geht nach folgendem Strickmuster vor. Wir nehmen uns die letzte Stelle vor, konkret an unserem letzten Beispiel die "9", und multiplizieren

Sie mit der Wertigkeit der Stelle. An der letzten Stelle ist dieser Wert dann 9 * 1 (Wertigkeit) = 9. Die nächste Stelle hat die Wertigkeit 16, wir multiplizieren also unsere Variable, die die Wertigkeit enthält wieder mit der Basis (16). Mit dem neuem Wert behandeln wir die nächste Stelle: 10 * 16 = 160.

Alle errechneten Zwischenergebnisse werden in einer separaten Summe addiert. Das ist genau der Weg, den wir "zu Fuß" selbst berechnet haben, mit dem Unterschied, daß hier jeder Schritt für unser Programm noch einmal in Schrittchen zerlegt werden muß. Sie brauchen nicht zu verstehen, warum das so gerechnet wird, da Sie doch C und nicht Mathematik lernen wollen.

9.3.1 Kleines Umrechenprogramm

Analysieren Sie das folgende Listing anhand der bereits gestellten Überlegungen! Ist es Ihnen klar? Wenn nicht, empfehle ich, zwischen einige Berechnungen printf-Funktionen einzuschieben, die den gerade aktuellen Wert einer oder mehrerer Variablen ausdrucken. Sie haben dadurch die wichtigsten Variablen immer im Blickfeld.

```
main()
{
  long basis, sammel, wertigkeit;
  int index, help;
  char zahl[100];

  printf("Bitte Basis eingeben!\n");
  scanf("%ld", &basis);
  printf("Bitte umzurechnende Zahl im %ld-System eingeben!\n", basis);
  scanf("%80s", zahl);
  sammel = 0;
  wertigkeit = 1;
  index = strlen(zahl) - 1; /* Neue Funktion */
  while( index >= 0)
    {
      help = zahl[index];
      if(help >= 'a')     /* Kleinbuchstabe? */
        help = help - 'a' + 10;
      else
```

```
        if(help >= 'A')   /* Grossbuchstabe? */
          help = help - 'A' + 10;
        else          /* wohl doch eine Ziffer! */
          help = help - '0';
      sammel = sammel + wertigkeit * help;
      index = index - 1;
      wertigkeit = wertigkeit * basis;
    }
  printf("%s(%ld) = %ld(10)\n", zahl, basis, sammel);
}
```

In obigen Programm machen wir nun schon regen Gebrauch von dem "neuen" Datentyp long. Dies macht sich dann auch bei den Steuerzeichen für die Ein- und Ausgabe dieser Varibalen bemerkbar, es taucht ein "%ld" auf. Auch in der scanf-Funktion, die eine Zahl als String einliest, finden wir eine Neuerung. Nach dem Prozentzeichen wurde eine "80" plaziert, danach folgt erst die eigentliche Formatanweisung "s" für den String. Dieser Wert zwischen Prozentzeichen und der Formatanweisung teilt der Funktion mit, wieviel Zeichen wir maximal benötigen. In diesem Fall kann der String also nicht länger als 80 Zeichen sein (+ 1 Endekennzeichen = 81). Ganz so, wie man es sich vorstellt läuft das aber nicht ab. Wir erhalten zwar maximal 80 Zeichen, der Benutzer kann aber ohne weiteres mehrere Zeilen vollschreiben. Dann erhalten wir davon nur die ersten 80 Zeichen.

Die Angabe einer maximalen Stellenanzahl ist natürlich auch bei den anderen Datentypen erlaubt (siehe Beispiel bei scanf). Als nächstes lernen wir eine neue Funktion kennen: strlen. Schon aus dem Namen kann man auf ihre Aufgabe schließen, sie liefert die Anzahl der Zeichen in einem String. Darin ist aber nicht das abschließende Nullbyte enthalten. Das Resultat wird wie bei einer Variablen durch Gleichheitszeichen zugewiesen. Den einzigen Parameter, den strlen benötigt, ist der zu untersuchende String. Wir hatten ja sogar schon einen kleinen Programmausschnitt vor uns, in dem genau dies, nämlich die Länge eines Strings zu berechnen, auch abgehandelt wurde. Besonders umfangreich ist diese Funktion also nicht.

In der folgenden while-Schleife "verarzten" wir dann den eingegebenen String. Um bei Berechnungen nicht jedesmal den Ausdruck "zahl[index]" zu verwenden, wird der dort befindliche Buchstabe erst einmal in die Variable "help" kopiert. Es müßte Ihnen bei genauen Hinschauen aufgefallen sein, daß "help" als Integervariable definiert ist und wir nun frech fröhlich versuchen dort ein Zeichen abzulegen. Aber was sind schon Zeichen und Buchstaben für den Computer? Im Grunde seines Herzens hält er den Zahlen die Treue. Alle Zeichen sind nichts als Zahlen für ihn, die genau wie z.B. das Alphabet in einer bestimmten Reihenfolge geordnet sind. Jeder Buchstabe hat eine charakteristische Kennzahl, genauso wie eine Ziffer oder ein Sonderzeichen. So sind "char"-Variablen nichts anderes als noch kleinere Integerspeicher, die je nach Compiler einen Wertebereich von -128 bis 127 oder von 0 bis 255 zulassen. Auf diese Besonderheiten, mit Zahlen und deren Codes zu rechnen, kommen wir gleich zurück.

Nachdem uns das Zeichen in "help" zur Verfügung steht, wird geprüft, ob es sich um einen Klein- oder Großbuchstaben handelt. Diese dürfen ja bei Zahlensystemen, deren Basis größer als 10 ist, als Ersatzziffern eingesetzt werden. Im Hexadezimalsystem benutzen wir dafür die Buchstaben A-F. Wenn feststeht, um welche Zeichengattung (Groß-, Kleinbuchstabe oder Ziffer) es sich handelt, errechnen wir daraus den wirklichen Wert. Denn '9' ist nicht gleich 9! Verwirrt? Dann klären wir das schnell. Sie wissen ja noch, wie ein Zeichen an "char"-Variablen zugewiesen wurde:

```
zeichen = '9';
```

9.4 Code eines Zeichens

Die '9' ist ein Zeichen, das die Ziffer neun darstellt. Dieses Zeichen besitzt aber auch einen speziellen Zeichencode. Bei der neun ist dies beispielsweise der Code mit dem Wert 57. Die Zuweisung

```
zeichen = 57;
```

ist dadurch mit der anderen Formulierung gleichbedeutend. Rechnen wir aber mit einer solchen Variablen, benötigen wir den Wert 9 und nicht den bislang gespeicherten Code 57. Davon ziehen wir erst einmal 48 ab, genau den Code, den das Zeichen '0' besitzt. Das ist ganz praktisch, alle Ziffern kommen hintereinader mit fortlaufenden Codes:

Code	Zeichen
48	0
49	1
50	2
51	3
...	
57	9

Bei Buchstaben aus dem Alphabet besteht auch so eine Ordnung. Hier beginnt die Tabelle bei 'A' mit dem Code 65, 'B' entspricht 66 usw. Die Kleinbuchstaben sind wiederum in einer separaten Liste zusammengefaßt. Der erste Wert dort ist 97 für das Zeichen 'a'. Betrachten wir uns einen Programmausschnitt:

```
char zeichen;

zeichen = 'B';
zeichen = zeichen - 'A' + 10;
```

Was steht nach Ablauf dieser Sequenz in der Variablen "zeichen"? Ein äquivalenter Teil taucht nämlich auch in dem Umrechnungsprogramm auf. Haben Sie das Ergebnis? 11 ist die richtige Antwort. In der letzten Zeile angekommen, enthält "zeichen" den Buchstaben 'B' mit dem Wert 66. Ziehen wir nun 'A' (65) ab, erhält man 1 plus 10 sind 11. Das ist genau der Wert, den der Buchstabe "B" im Hexadizimalsystem darstellt.

Wollten wir einen Großbuchstaben in einen entsprechenden Kleinbuchstaben umwandeln, so genügt folgende Zeile:

```
zeichen = zeichen - 'A' + 'a';
```

Das sieht doch übersichtlicher aus als

```
zeichen = zeichen - 65 + 97;
```

oder

```
zeichen = zeichen + 32;
```

Die Codes haben die Eigenschaft, auf (fast) jedem Rechner gleich zu sein, da es hierfür sogar eine Norm gibt. Vielleicht ist Ihnen der ASCII-Code ja schon ein Begriff, so nennt man diese Tabelle, die jedem Zeichen einen Code zuschreibt.

Damit Sie einen Überblick über die ASCII-Codes bekommen, benutzen wir ein kleines Programm, das zu jedem Code (32 - 127 und 160 - 255) das dazugehörige Zeichen ausgibt. Die Codes 0 - 31 und 128 - 159 wurden ausgelassen, weil sie entweder eine besondere Aufgabe haben (13 entspricht '\n') oder überhaupt nichts auf dem Bildschirm produzieren.

```
main()
{
  int i;
  printf("\n\n");
  for(i = 32; i <= 127; i = i + 1)
    printf("\t%-3d %c", i, i);
  for(i = 160; i <= 255; i = i + 1)
    printf("%-3d %c \t", i, i);
  printf("\n\n");
}
```

Wozu brauchen wir in der Praxis aber die Codes? Ein Text wird so, wie er ausgegeben werden soll, zwischen die Anführungszeichen der printf-Funktion geschrieben.

9.4.1 Der Backslash hilft

Möchten Sie ein Zeichen ausgeben, welches nicht direkt oder mit Baskslash zu erreichen ist, so kann man es über den Code des Zeichens ansprechen. Vor den Code braucht man lediglich den Backshlash zu stellen. Der Compiler erkennt daran, daß er diese Zeichenkombination aus Backslash und den einzenen Ziffern der Codezahl durch ein einziges Zeichen ersetzen soll. Die Möglichkeit hat nur einen kleinen Haken, die Zahl kann nicht dezimal, sondern muß oktal eingegeben werden. So kann beispielsweise das "±"-Zeichen, ein Spezialzeichen des AMIGA, mit dem Zeichencode 177 mit dem Befehl

```
printf("\261");
```

auf den Bildschirm transportiert werden. Die Zahl 177 im Dezimalsystem entspricht 261 im Oktalsystem. Mit einem kleinen Trick kann die Umrechnung aber umgangen werden:

Anstelle von

```
printf("\261");
```

schreiben Sie

```
printf("%c", 177);
```

Sie benutzen also nicht das Zeichen mit direkt innerhalb der zeichenkette, sondern übergeben der Funktion printf, durch das "%c"-Zeichen kenntlich gemacht, direkt diese gewünschte Information in Form des Zeichencodes.

```
printf("Das Ergebnis ist \2611.\n");

printf("Das Ergebnis ist %c1.\n", 177);
```

Das Steuerzeichen "%c" ermöglicht ja die Ausgabe eines einzelnen Zeichens über Angabe des Codes, auch wenn Sie hierfür einen Integerwert übergeben. Sie sehen, es gibt kein Problem, das sich nicht geschickt umgehen läßt.

9.4.2 In die andere Richtung

Für den, der aber nun erst recht darauf besteht, die Oktalzahlen aus den Dezimalzahlen zu berechnen ist hier ein fertiges Programm abgedruckt, das diese Aufgabe für Sie erledigt. Es ist quasi die Umkehrung des vorherigen Umwandlungsprogramms, das Zahlen ins Dezimalsystem übertrug.

```
main()
{
  long basis, zahl, help, rest;
  int index;
  char resultat[260];

  printf("Bitte Basis eingeben!\n");
  scanf("%ld", &basis);
  printf("Bitte umzurechnende Zahl im Dezimal-System eingeben!\n");
  scanf("%ld", &zahl);
  index = 0;
  for(rest = zahl; rest > 0; rest = rest / basis)
    {
      help = rest % basis; /* Rest der Division */
      if(help > 9)  /* Ersatzweise muss Buchstabe herhalten */
        resultat[index] = help + 'A' - 10;
      else
        resultat[index] = help + '0';
      index = index + 1;
    }
  printf("%ld(10) = ", zahl);
  index = index - 1; /* Letzter Eintrag ist noch unbenutzt */
  while(index >= 0)
    {
      printf("%c", resultat[index]);
      index = index - 1;
    }
  printf("(%ld)\n", basis);
}
```

So, damit hätten wir die wichtigsten Dinge zum Strings und Zeichen abgehandelt. Nehmen wir uns einem völlig neuem Thema an.

10. Der Präprozessor

Dieser Begriff fiel bisher nur in der Beschreibung des C-Compilers. Was man sich darunter vorstellen und wie man ihn benutzen kann, erfahren Sie jetzt.

Der Präprozessor ist ein Programmteil des Compilers, der den Text zuerst bearbeitet. Er nimmt ihn so entgegen, wie wir ihn erstellt haben. Nun gibt es ein paar Spezialanweisungen, die den Präprozessor veranlassen, einige Änderungen am Programmtext vorzunehmen. Nachdem der Präprozessor seine Arbeit beendet hat, erhält der für die Übersetzung zuständige Teil des Compilers die neue Version unseres Programms. Die kann dann schon ganz anders aussehen, als das von uns erstellte Listing.

Damit man die für den Präprozessor bestimmten Befehle von den C-Befehlen unterscheiden kann, gibt es zwei wichtige Richtlinien:

1.) Alle Befehle beginnen mit dem Zeichen "#"
2.) Alle Befehle beginnen in der ersten Spalte

10.1 #define

Nehmen wir uns zuerst den wohl wichtigsten und am meisten genutzten Präprozessor-Befehl vor: #define. Mit "#define" definiert man für eine bestimmte Zeichenfolge einen Ersatztext. Vom Präprozessor werden die beiden Texte dann ausgetauscht. Was so einfach klingt, kann auch sehr kompliziert werden, ist aber auch eine der Stärken von C. Überlegen wir uns, wozu wir überhaupt einen Textersatz brauchen. Nehmen wir an, wir benutzen eine Konstante bei Berechnungen. Konkret: Bei einem kleinen Mehrwertsteuerprogramm taucht des öfteren ein bestimmter Prozentsatz z.B. 14% auf. Wenn dieser Wert im Programm vielleicht 10-20mal verwendet wird, ist eine nachträgliche Änderung, wenn die Mehrwertsteuer erhöht wird, fehleranfällig. Man könnte einen falschen Wert ändern, beispielsweise, weil irgendwo zufällig die Zahl 14 auftaucht oder auch einfach einen Eintrag übersehen. Einfacher und auch leserlicher gestaltet

ist das Programm bei Verwendung von Defines mit dem "#define"-Befehl. Eine entsprechende Anwendung sähe so aus:

```
#define MWST 14
```

Ab dieser Zeile kann man nun den Text "MWST" verwenden, der vom Präprozessor durch den Text "14" ersetzt wird. Es wäre also auch folgende Zeile denkbar:

```
printf("MWST-Satz %d", MWST);
```

Der Präprozessor hinterläßt dem Übersetzer folgende korrigierte Zeile:

```
printf("MWST-Satz %d", 14);
```

Innerhalb der Anführungsstriche hat sich nichts geändert. Das ist auch gut so, sonst wäre es ja unmöglich, eine derartige Zeichenkette wie "MWST" auf dem Bildschirm auszugeben. Alles, was innerhalb von Anführungszeichen steht, ist für den Präprozessor tabu.

Defines sind praktisch aus keinem größeren Programm mehr herauszudenken. Ich hoffe, daß auch das folgende kleine, recht sinnlose Programm, aber auch die Verwendungsweise von Defines deutlich macht. Was das Programm produziert, kann man ohne Probleme dem Listing entnehmen.

```
#define ANFANG 1
#define ENDE 100
#define SCHRITTWEITE 2

main()
{
  int i;

  printf("\n");
  for(i = ANFANG; i <= ENDE; i = i + SCHRITTWEITE)
    printf("%5d", i);
  for(i = ENDE; i >= ANFANG; i = i - SCHRITTWEITE)
    printf("%5d", i);
```

```
    printf("\n");
}
```

Aber auch bei kleineren Programmen kann die Verwendung der "#define"-Anweisung den Lesefluß erhöhen. Ein Beispiel ist die Markierung des String-Endes mit einem Null-Eintrag. Dieses Nullbyte wird auch als "End Of String" (Ende des Strings) bezeichnet. Unter der Abkürzung EOS findet man eines der meistverwendeten Defines. Die Definition sieht so aus:

```
#define EOS '\0'
```

Das ist korrekter, als nur eine 0 anzugeben. Wir betrachten doch die Einträge eines Strings als einzelne Zeichen. Deshalb ist es auch guter C-Stil, dem Datentyp entsprechende Zuweisungen zu verwenden. Durch die einfachen Anführungszeichen wird dem Compiler schon einmal mitgeteilt, daß wir hier ein einzelnes Zeichen zusammengesetzt haben. Der Slash gefolgt von dem Oktalwert gibt dann den Code des Zeichens an. Sie erinnern sich doch noch an das Kapitel zuvor?!

Die Zahl Null im Oktalsystem ist aber ebenso wie im Dezimal- und allen anderen Zahlensystemen ebenfalls Null. Eine Umrechnung quält uns in diesem Fall nicht. Ob wir nun das Zeichen mit dem Code null, oder direkt den Code (eben null) bei der Zuweisung benutzen, ist gehupft wie gesprungen. Wenn wir in Zukunft Programme schreiben, die sich um Strings kümmern, sollte in einer der ersten Zeile die Definition von EOS stehen.

Wenn Sie nun glauben, das Thema "#define" wäre damit abgeschlossen, haben Sie sich geirrt. Die vielfältigen Möglichkeiten, die einem durch die "#define"-Anweisung geöffnet werden, werden wir in einem separaten Kapitel noch einmal ausführlich beleuchten.

10.2 #include

Sehen wir uns nun einen anderen überaus wichtigen Präprozessor-Befehl an. Mit dem "#include"-Kommando läßt sich während des Compilierens ein File zur aktuellen Datei hinzuladen. Das kann man sich so vorstellen, als würde man im Editor mit dem Befehl "File hinzufügen" (ESC I F beim ED) die angegebene Datei in die eigene Source-Datei hinzuladen und komplett abspeichern. Der Compiler macht beim Übersetzen keinen Unterschied, wo irgendwelche Definitionen vorgenommen wurde, für ihn existiert nur eine Datei. Deshalb eignet sich dieser Präprozessor-Befehl hervorragend, um ganze Kolonnen von Defines in eigene Programmme einzubeziehen. Nehmen wir an, Sie würden folgende Defines unter dem Namen "def_neu.h" auf Diskette abspeichern.

```
#define EOS '\0'
#define MAXLEN 81
#define EOF -1
```

Haben Sie nun ein Programm geschrieben, das einige der aufgeführen Defines benutzt, so brauchen Sie nicht jedesmal neu alle Definitionen einzutippen, es genügt die Datei zu "includen".

```
#include "def_neu.h"
```

Die Endung ".h" dient, ebenso wie ".C oder ".O" zur Identifizierung der Datei. Das "H" steht für Header (engl. Kopf) und beschreibt seine Funktion korrekt. Das Einbinden solcher Datei wird immer am Anfang einer Datei praktiziert, um sicher zu gehen, daß an jeder Stelle im Programmlisting auch sämtliche Defines zur Verfügung stehen. Es ist zwar nicht vorgeschrieben, Sie sollten sich aber daran orientieren. Die Präprozessor-Befehle können überall in einer Datei auftauchen, da wir aber kein Freistil-C wollen, sollten wir diese Möglichkeiten außer acht lassen.

Den Dateinamen haben wir in Gänsefüßchen gefaßt. Das ist eine Variante! In diesem Fall sucht der Compiler in dem Unterverzeichnis, in dem sich auch der Sourcecode befindet. Eine andere

Variation ist durch die Verwendung des Kleiner- und des Größerzeichens zu produzieren.

```
#include <def_neu.h>
```

Jetzt vermutet der Compiler die Datei "def_neu.h" in einem speziellen Ordner, in dem sämtliche ".H"-Datei zu finden sind. Der Weg bis in dieses Subdirectory (Unterverzeichnis) wird dem Compiler in der Regel beim Start mitgeteilt. Es gibt schon eine Reihe solcher Dateien, die nur darauf warten, von Ihnen benutzt zu werden. Eine der beliebtesten Dateien dieser Art ist unter dem Namen "stdio.h" auf der Diskette zu finden. Beim Lattice-C müssen Sie sich allerdings zuerst über einen längeren Pfad bis dorthin vortasten. Wenn Sie mal einen Blick in dieses File werfen wollen, geben Sie zum Aufruf des ED folgende Zeile ein:

```
ed df0:include/lattice/stdio.h
```

Hier wird vorausgesetzt, daß sich die Compilerdiskette im Laufwerk df0: befindet und Sie natürlich den Lattice-Compiler benutzen. Aber auch wenn Sie einen anderen Compiler benutzen, dürfte es kein Problem sein, diese Datei ausfindig zu machen.

Wenn Sie ein bißchen durch die Datei blättern, bitte fallen Sie nicht gleich vom Hocker. Was dort definiert ist, brauchen und können Sie noch gar nicht verstehen. Wir werden aber später bestimmt auf diese Definitionen zurückgreifen, so daß es sinnvoll ist, jetzt zumindest ansatzweise zu wissen, wo die Informationen herkommen.

11. Abkürzungen

Langsam, aber sicher nähern wir uns den Spezialitäten von C. C hat ein Faible für schreibfaule Programmierer. Fangen wir mit den einfachsten Sparmaßnahmen an: den Abkürzungen bei arithmetischen Operationen. Wie oft haben wir beispielsweise schon solche Zeilen geschrieben:

```
zahl = zahl * 4;
```

Was könnte man hier noch sparen? Es taucht zweimal die Variable "zahl" auf, also kann gekürzt werden:

```
zahl *= 4;
```

Jedesmal, wenn die gleiche Variable bei der Berechnung und wieder zum Speichern des Ergebnises verwendet wird, kann eine solche Kurzform benutzt werden. Der Operator wandert auf die linke Seite des Zuweisungszeichens. Der Teil, mit dem in diesem Fall die Varibale "zahl" multipliziert werden soll bleibt auf der rechten Seite bestehen. Wichtig und effektiv wird die Benutzung solcher Formulierungen bei langen Variablennamen. Tippfehler lassen sich vermeiden:

```
eingabe_des_benutzers[index] += '0';
```

entspricht

```
eingabe_des_benutzers[index] = eingabe_des_benutzers[index] + '0';
```

Stellen Sie sich einmal vor, Sie müßten diesen Namen tatsächlich zweimal hinschreiben! Ein weiterer Vorteil, der nicht sofort ersichtlich ist, ist der Geschwindigkeitsvorteil des erzeugten Objektcodes. Der Compiler weiß bei der abgekürzten Schreibweise, welchen Wert wir benutzen und daß dort auch das Ergebnis wieder abzuliefern ist. Er kann durch diese Vereinfachung eine Menge an unnötigen Berechnungen einsparen.

Alle arithmetischen Operationen und Verknüpfungen lassen sich in dieser Weise verkürzen:

+=
-=
*=
/=
%=
usw.

Sehen wir uns den folgenden Ausdruck an!

```
wert = wert * (5 + zahl);
```

Die Zeile ist bereits so schön geordnet, daß sofort auffällt, welcher der beiden Operatoren für die Abkürzung in Frage kommt. Es ist das Malzeichen, so daß vorerst diese Ziele entsteht

```
wert *= (5 + zahl);
```

Da generell bei solchen Abkürzungen zuerst die rechte Seite vom Gleichheitszeichen berechnet wird, benötigen wir keine Klammern.

```
wert *= 5 + zahl;
```

Das gleiche rückwärts gedacht! Wir dürfen nicht einfach den Operator und die genannte Variable an den übrigen Term anhängen, sondern müssen vorher Klammern setzen:

```
var *= zahl1 - zahl2;
```

entspricht

```
var = (zahl1 - zahl2) * var;
```

11.1 Inkrement und Dekrement

Das Minimieren der Tipparbeit geht noch einen Schritt weiter. Für zwei Spezialfälle stehen auch zwei dafür vorgesehene Operartoren bereit. In dem Fall, in dem wir eine Variable um den Wert 1 erhöhen oder verringern kann man wahlweise auch die Operatoren "++" und "--" benutzen. Der "++"-Operator erhöht die angegebene Variable um eins und heißt deshalb auch Inkrementoperator. Umgekehrt agiert der Dekrementoperator "--". Diese Operatoren muß man in Natura erleben:

```
main()
{
  int i;
  i = 1;
  while(i++ < 100)
    printf("%d ", i);
}
```

So kurz das Programm auch ist, es ist tückisch! Bis zur while-Schleife ist alles klar, "i" enthält den Wert Null. Jetzt kommt der Ausdruck

```
i++ < 100
```

zur Bearbeitung. Zuerst prüft der Rechner, ob i kleiner als 100 ist. Anschließend erhöht sich der Wert von i um 1, egal, wie der Test ausgefallen ist. Das entspricht den beiden einzelnen Befehlen

```
if(i < 100)
  bedingung = 1;
else
  bedingung = 0;
i = i + 1;
while (bedingung)
  ...
```

mit dem Unterschied, daß alle hier vor while aufgelisteten Befehle innerhalb der runden Klammern ausgeführt werden. Das hätte man dem kleinen "++"-Operator gar nicht zugetraut. Es kommt aber noch besser! Sie können den Inkrement- und den

Dekrementoperator vor und hinter der Variablen plazieren. Egal ist das aber nicht. Das verdeutlichen wir uns an einem einfachen Beispiel:

```
i = j++;
```

Unter der Annahme, "j" enthielte den Wert 3, bekommt "i" ebenfalls 3 zugewiesen. Anschließend erhöht sich der Wert von "j" um eins auf 4. Im Gegensatz dazu sehen wir uns die Zeile an, in der der Operator auf der anderen Seite steht.

```
i = ++j;
```

Bei gleichen Voraussetzungen wird zuerst der Inhalt von "j" auf 4 inkrementiert und danach erst der Variablen "i" zugewiesen. Beide Variablen enthalten also jetzt 4.

Merken wir uns: Steht einer der Operatoren "++" oder "--" vor der Variablen, so wird zuerst der Variableninhalt verändert, bevor er zu weiteren Untersuchungen verwendet wird. Steht der Operator hinter der Variablen, wird zuerst der augenblickliche Wert benutzt, und anschließend erfolgt die Inkrementierung oder Dekrementierung. Es ist wichtig, sich den kleinen, aber entscheidenen Unterschied zu merken. Sehen Sie sich bitte die Ausgabe des obigen Programms auf dem Bildschirm an. Die erste Zahl, die dort erscheint, ist die zwei. Das ist auch klar, denn der Startwert von "i" war eins, der bereits innerhalb des Schleifenkopfes von while inkrementiert wurde. Deshalb enthält "i" zu dem Zeitpunkt, zu dem der printf-Aufruf erfolgt, schon den Inhalt 2.

Apropos: Auch diese Operatoren helfen, schnelle und kompakte Programme zu schreiben. Sie sind noch effizienter als die bereits gelobten Abkürzungen mit dem Gleichheitszeichen.

11.2 Initialisierung, Definition, Deklaration

Die drei Begriffe sind sehr wichtig für C-Programmierer und dürfen keinesfalls über einen Haufen geworfen werden. Fangen wir bei der Initialisierung an. Sie beschreibt die erste Zuweisung einer Variablen mit einem Wert. Ab dieser Stelle wissen wir dann mit Sicherheit, was in der Variablen abgelegt ist. Bevor man allerdings die Variable benutzen, beziehungsweise initialisieren kann, muß sie dem Programm erst einmal bekannt sein. Sie muß entweder definiert und/oder deklariert sein. Die Definition kennen wir aus Zeilen wie

```
int index;
char string[80];
```

Sobald das Programm an dieser Stelle anlangt, kennt es die Variablen und stellt für sie den nötigen Speicherplatz bereit. Ein Integerwert benötigt in der Regel 2 Bytes, der oben angegebene String 80 Bytes, da jedes "char"-Element 1 Byte beansprucht. Man kann aber auch Funktionen definieren (Wir haben uns bislang auf eine einzige Definition, die von main beschränkt). Das ist dann auch der Unterschied zur Deklaration. Deklariert man eine Variable oder auch eine Funktion, so teilt man dem Programm lediglich mit, daß eine solche Variable oder Funktion irgendwo definiert wurde. Deshalb wird für sie auch kein Speicher zur Verfügung gestellt.

Wenn wir gerade bei diesem Thema sind, wieder ein Tip zum Fingerschonen. Variablen können bereits bei ihrer Definition initialisiert werden, das erspart eine Zeile.

```
int index = 0;
```

Es kommt noch besser, die Initialisierung beschränkt sich nicht nur auf Konstanten. Vielmehr können Sie einen beliebigen Ausdruck der gerade definierten Variablen zuweisen. So läßt sich die Länge eines Strings, die man mittels der Funktion strlen ermittelt, auch bei der Definition zur Initialisierung verwenden.

```
int ende = strlen(string) - 1;
```

Wobei string vorher definiert sein muß!

Man kann natürlich alles übertreiben, aber der Phantasie sind halt keine Grenzen gesetzt.

```
long mitte = 4*((strlen(string1)+1)/2+1)-strlen(string2)/3;
```

Haben Sie ein größeres Programm in mehreren Modulen (Dateien) geschrieben, so braucht eine Variable, die in allen Modulen benutzt wird, ja nur einmal den benötigten Speicherplatz. In einer Datei findet man die Definition, in allen anderen nur die entsprechende Deklaration. Die Deklaration führt man mit dem C-Wort "extern" durch. Der Compiler weiß dann, daß Sie, ordentlich wie Sie sind, in einer anderen Datei den Speicherplatz reserviert haben, und er sich auf Sie verlassen kann. Sollte dem nicht so sein, wird Ihnen der Linker die Meinung zu dem Thema sagen.

Beispiel:

```
extern char pass_wort[80];
extern int fehler_nr;
```

Wie Sie sehen, müssen Sie auch den Datentyp angeben. Dann hat der Compiler alle für ihn nötigen Information über die Variable. Die Deklaration einer Funktion läuft übrigens analog dazu ab:

```
extern long atol();
```

Sollte die Funktion jedoch in der gleichen Datei noch definiert werden, kann man das "extern" weglassen. Die Deklaration ist aber trotzdem nötig, da der Compiler ja erst am Ende der Datei alle Funktionsnamen und deren Datentypen kennt.

11.3 Mehrfachzuweisungen und Wert eines Ausdrucks in C

Tipparbeit kann man sich ebenfalls bei der Mehrfachzuweisung sparen. Wollen Sie mehreren Variablen den gleichen Wert zukommen lassen, benötigten Sie bislang für jede Variable eine einzelne Zuweisung, wobei jedesmal der gleiche Wert angegeben wird:

```
anfang = 0;
summe = 0;
```

Es genügt aber folgende Zeile:

```
anfang = summe = 0;
```

Die Zuweisung erfolgt von rechts nach links. Zuerst wird "summe" 0 zugewiesen, dann erhält "anfang" den Inhalt von "summe", und der ist ja 0. Ein Term mit noch mehr gleichzeitigen Zuweisungen, wie

```
a = b = c = d = 2;
```

könnte mit Klammern umgeben werden, die zwar nicht notwendig sind, aber die Reihenfolge der Abarbeitung verdeutlichen:

```
a = (b = (c = (d = 2)));
```

Einzeln ausgedrückt entspräche dieser Ausdruck:

```
d = 2;
c = d;
b = c;
a = b;
```

Die Mehrfachzuweisung ist möglich, da jeder Ausdruck einen Wert besitzt (das Ergebnis der zuletzt ausgeführten Operation), z.B. ist der Wert von (d = 2) 2, von (index = strlen(string)) strlen(string). Außer bei umfangreichen Initialisierungen von Variablen kann man den Wert eines Ausdrucks fast überall gebrauchen. Und hier zeigt sich dann auch, wer sich in C aus-

kennt. Man kann fast sagen, je kürzer die Formulierung, desto professioneller.

An Hand von Beispielen läßt sich das wohl schnell deutlich machen. Hier nun weitere Werte von Ausdrücken

```
(2)                2
(a)                a
(a *= 3)           a*3
(a=(b=(a+2)-3))    a-1
```

Das letzte Beispiel muß man schon in die Bestandteile zerlegen, will man zu dem gleichen Ergebnis gelangen.

```
(a=(b=(a+2)-3))
(a=(b= a-1 ))
(a=( a-1 ))
( a-1 )
```

Natürlich können Sie den Vorzug der Mehrfachzuweisung bereits bei der Definition und Initalisierung nutzen. So ist folgende Zeile durchaus zugelassen:

```
int anfang = wert = 0;
```

Die Variable "wert" muß allerdings schon zuvor definiert und, das ist besonders wichtig, initialisiert worden sein, was ja genau hier schon der Fall ist.

12. Funktionen

Wie schon in der Einleitung gesagt, besteht ein C-Programm aus einer Menge verschiedener Funktionen. Wir haben aber stets nur eine einzige definiert, nämlich die main-Funktion. Es wird also Zeit, daß wir Programme schreiben, die mehrere, von uns selbst entwickelte Funktionen enthalten.

Erst zum formalen Aufbau einer Funktionsdefinition: Zuerst wird der Funktionsname angegeben, vor dem der Datentyp angegeben ist, den diese Funktion zurückliefert. Der Name muß den gewohnten Regeln für Variablennamen entsprechen. Nach dem Namen folgen runde Klammern, in die die Übergabeparameter eingeschlossen sind. Wenn es keine solchen Werte gibt, wie z.B. bei der main-Funktion, können wir auch keine angeben. Logisch! Falls wir aber solche Parameter erwarten, müssen diese Varibalen im folgenden noch deklariert (Sie sind im Bilde!) werden. Die Werte sind wichtig, da die meisten Funktionen Informationen von anderen Funktionen erhalten, die sie dann verarbeiten. Anschließend kommen noch die in geschweiften Klammern eingeschlossenen und auszuführenden Befehle. Führen wir uns die wie selbstverständlich benutzte main-Funktion vor Augen.

```
main()
{
  ...
}
```

Das erste, was wir laut unserer Vorschrift hier anzutreffen haben, ist der Datentyp, den das Programm zurückgibt. Da die main-Funktion dem aufrufenden Programm keine Werte liefert, steht kein Datentyp voran. Nun folgt der Funktionsname, hier "main", gefolgt von runden Klammern. Da wir keine Werte von außen bekommen (bekommen wollen), bleibt der Platz innerhalb der Klammern frei. Dementsprechend entfällt auch die Deklaration möglicher Übergabeparameter. Dann folgen alle auszuführenden Instruktionen innerhalb der geschweiften Klammer, also bislang unser gesamtes Programm.

12.1 Funktionen mit Parametern

Gehen wir jetzt aber einen Schritt weiter und zerlegen das Programm in einzelne Teilaufgaben. Für jede Teilaufgabe schreiben wir dann eine kleine Funktion. Nehmen wir das Beispiel "Quadratzahl", das keine große mathematische Vorbildung verlangt.

```
double quadrat(x)
float x;
{
  double q_zahl;

  printf("Ich berechne das Quadrat von %f\n", x);
  q_zahl = x * x;
  return q_zahl;
}
```

Diese Funktion definiert eine Funktion mit Namen "quadrat", die selbst wiederum einen double-Wert an das aufrufende Programm zurückgibt. Als Übergabeparameter wird ein float-Wert benötigt, der in dieser Funktion "x" getauft wird. Am Ende der Quadratroutine erkennen Sie ein neues C-Wort, den return-Befehl. Er liefert das gewünschte Ergebnis in dem angegebenen Datentyp an den Aufrufer. Dadurch ist zugleich auch die Funktion beendet.

Wichtig ist, daß nach dem Funktionsnamen kein Semikolon kommt, nach den einzelnen Parameterdeklarationen aber. Dadurch kann nämlich eine Funktionsdefinition (ohne Semikolon) von einem Funktionsaufruf (mit Semikolon) unterschieden werden. Im aufrufenden Programm (z.B. Hauptprogramm), kann dann folgende Zeile vorkommen:

```
double quadrat();

main()
{
  float wert = 3.0;
  double ergebnis;
  ...
  ergebnis = quadrat(wert);
}
```

Wie Sie erkennen können, brauchen die Namen der Übergabeparameter bei der aufrufenden Funktion nicht mit denen der aufgerufenen Funktion identisch zu sein. Lediglich die Datentypen der Parameter müssen übereinstimmen. Beachten Sie auch die Zeile, in der die Funktion "quadrat" als Funktion deklariert wird, die double-Werte zurückgibt.

Da C für (fast) alles Möglichkeiten für Schreibfaule bietet, kann die Deklaration auch entfallen, wenn Integerwerte zurückgeliefert werden. Das gleiche gilt auch bei der Definition einer Funktion. Sollte die Funktion int-Werte liefern, dann braucht kein Datentyp vor den Funktionsnamen gestellt werden. Dies ist aber nur bei dem Datentyp "int" möglich, alle anderen Typen müssen vor ihrer Benutzung deklariert und bei der Definition mit dem richtigen Datentypen versorgt werden. Sollten sich diese Daten widersprechen, beispielsweise weil die Deklaration einer double-Funktion vergessen wurde, so erhalten Sie in der Regel total unsinnnige Ergebnisse zurück. C läßt halt dem Programmierer freien Lauf, was aber auch die entsprechende Gefahren in sich birgt.

12.2 Funktionen ohne Rückgabewert

Es können natürlich auch Funktionen auftauchen, die keinen Wert zurückliefern. Diese Funktionen können als "void" deklariert werden, falls Ihr Compiler dieses C-Wort implementiert hat. Dadurch kann vielleicht die Geschwindigkeit ein wenig erhöht werden, da keine Parameter mehr für die aufrufende Funktion bereit gestellt werden müssen, die diese dann doch nicht verwendet. Aber auch das kann weggelassen werden, weshalb der Typ "void" in einigen C-Compilern fehlt.

```
void verschluessel(string)     /* Ohne Rueckgabewert: void */
char string[80];
{
  int i;
```

```
  for (i = 0; string[i] > 0; i++)
    printf("%c", string[i] + 1);  /* Aus 'A' mach 'B' */
}


void main()
{
  char text[81];
  void verschluessel();

  printf("\n\nBitte Text eintippen!\n");
  scanf("%80s", text);
  verschluessel(text);
  printf("\nIm Original heißt das %s\n", text);
}
```

Wir rufen unsere selbstdefinierte Funktion genauso auf, wie es auch mit Routinen aus den Libraries gemacht wird. In diesem Beispiel steht die main-Funktion am Ende der Datei. In ihr wird die Routine namens "verschluessel" als Funktion deklariert, die "void", also nichts zurückliefert. Das ist wichtig, da sich die Angaben bei der Definition und bei der Verwendung in main sonst widersprechen. Wurde die Funktion nämlich nicht deklariert, so nimmt der Compiler an, daß es sich um int-Objekte handelt, die zurückgeliefert werden. Und das ist etwas anderes als "nichts".

12.3 Eigene Funktionen

Eine andere Funktion, die kein Ergebnis zurückliefert, ist strcpy. Diese Routine dient zum Kopieren von Strings und ist für den täglichen Umgang mit Zeichenketten unentbehrlich. Obwohl sie mit jedem Compiler in der Libraray mitgeliefert wird, ist es doch mal interessant zu sehen, wie man so etwas programmiert.

12.3.1 strcpy-Version 1

Überlegen Sie, welche Parameter eine solche Kopierfunktion benötigt! Das ist relativ einfach: Zwei Strings, wobei der eine in den anderen kopiert werden soll. Im Unterschied zum vorherigen Beispiel wissen wir allerdings nicht, wie viele Einträge die einzelnen Strings besitzen. Aber das ist auch gar nicht nötig, diese Angabe können wir ruhig unter den Tisch fallen lassen. Es reicht dem Compiler, wenn er weiß, daß er einen String erhält.

In der Routine selbst testen wir mit einem Index alle Einträge durch und kopieren sie, bis wir an das Ende, das sogenannte EOS-Zeichen, gelangen. Das Endekennzeichen muß natürlich ebenfalls noch übertragen werden.

```
#define EOS '\0'

strcpy(nach,von)
char nach[], von[];
{
  int i = 0;
  while((nach[i] = von[i]) != EOS)
  i++;
}
```

Der Funktion kann der Platzbedarf dieses Arrays egal sein, da sie keinen Speicher bereitstellen muß. Es wird direkt auf die in der aufrufenden Funktion definierten Einträge zugegriffen. Außerdem können die Strings ohnehin unterschiedliche Längen haben.

Na, was halten Sie denn von der Abbruchbedingung in der while-Schleife? Wieder ein typischer Fall von "Wert eines Ausdrucks"! Durch die Klammersetzung ist die Bearbeitungweise klar zu erkennen. Zuerst erfolgt die Zuweisung des Eintrags "von[i]" an "nach[i]". Der Klammerausdruck besitzt dann ebenfalls den Wert "von[i]", also das Zeichen, welches gerade kopiert wurde. Das wird nun mit dem Endekennzeichen verglichen. Wurde gerade EOS kopiert, so ist die Bedingung nicht mehr er-

füllt, und die Schleife wird verlassen. Andernfalls inkrementiert man den augenblicklichen Index und bleibt in der Schleife.

Sie sehen, der eigentliche Schleifenrumpf spielt nur eine kleine Nebenrolle, die Hauptsache findet in der Abbruchbedingung statt. Kopieren wir mit dieser Funktion ein bißchen herum. Beachten Sie immer, daß der String, in dem die Kopie abgelegt werden soll, zuerst kommt.

```
#define EOS '\0'
#define MAXLEN 81


strcpy(nach,von)
char nach[], von[];
{
  int i = 0;
  while((nach[i] = von[i]) != EOS)
    i++;
}


main()
{
  char s1[MAXLEN], s2[MAXLEN], s3[MAXLEN];

  printf("Ihren Namen bitte\n");
  scanf("%40s", s1);
  strcpy(s3, s1);
  strcpy(s2, "TEXT IN S2");
  printf("Also %s, in s2 befindet sich \"%s\".", s1, s2);
  printf(" Ich hoffe %s, daß das klar ist!\n", s3);
}
```

Die strcpy-Funktion kann zur Initialisierung von Strings herangezogen werden, da der folgende Ausdruck in C nicht zugelassen ist.

Falsch:

```
main()
{
  char text[20] = "Dideldum!";
  ...
}
```

Richtig:

```
main()
{
  char text[20];
  strcpy(text, "Dideldum!");
  ...
}
```

Hiermit wird der gesamte String in die Variable "text" kopiert.

12.3.2 strlen

Eine weitere interessante Routine zur Stringbehandlung ist die strlen-Funktion, die wir bereits benutzt haben. Sie ist zwar noch einfacher zu schreiben, liefert dafür aber einen Wert zurück. Die übergebene Länge des Strings ist eine ganze Zahl, sollte also ein Integerwert sein.

```
strlen(string)
char string[];
{
  int i = 0;
  while(string[i])
  i++;
  return(i);
}
```

Eine nette kleine und besonders kurze Funktion, nicht? Der Ausdruck "string[i]" ist immer der Inhalt dieses Elementes. Das bedeutet, der Ausdruck ist nur dann 0 (falsch), wenn auch das Endezeichen '\0' (EOS) erreicht ist. Der Index, der auch gleichzeitig der Länge der Zeichenkette entspricht, wird der aufrufen-

den Funktion als Integerwert mittels return-Anweisung übermittelt. Diese Funktion brauchen Sie nicht in der aufrufenden Funktion zu deklarieren, weil ja ein int-Wert zurückgegeben wird.

Sollten Daten mit dem return-Befehl übergeben werden, sollten Sie sicherstellen, daß der Wert auch den korrekten Datentyp aufweist. Wenn in der Funktionsdefinition angegeben ist, daß die Routine beispielsweise ein char-Element liefert, sollte auch hinter "return" eine Variable oder Konstante vom Typ "char" vorzufinden sein. Einige Compiler lassen sich solche Schluderigkeit nicht gefallen und meckern drauflos. Anderen ist das egal, die wandeln von sich aus das Ergebnis in den bei der Definition angegebenen Datentyp um. Machen Sie es am besten gleich richtig!

13. Arrays

Strings benutzen wir die ganze Zeit, als hätten wir einen eigenen Datentyp vor uns. Als einziger Unterschied fällt vielleicht die Definition einer Zeichenkette auf. Es ist aber bereits angeklungen, daß wir es beim String mit mehreren "char"-Einträgen zu tun haben. Eine solche Kette gleicher Objekte nennt man Array, ist aber nicht nur auf char-Objekte beschränkt. Wer hindert uns denn, anstelle von "char" den Datentyp "int" oder "float" dort zu postieren?

Alle elementaren Datentypen können auch in einem Array angeordnet werden. Über einen einzigen Namen sind so eine Vielzahl gleichartiger Variablen erreichbar, wobei allerdings auch auf jedes einzelne Elemente über eine Nummer, dem sogenannten Index, zugegriffen werden kann. Eine Definition eines long-Arrays unterscheidet sich kaum von der bislang geübten Praxis einer Stringdefinition.

```
long werte[20];
```

Die Zeile legt fest, daß 20 Elemente vom Typ long für den Namen "werte" reserviert werden. Um das Ende einer Zeichenkette anzuzeigen, erhält ja der letzte Eintrag den Wert 0, d.h. ihm wird das Zeichen '\0' zugeordnet. Deshalb ist bei der Definition eines Zeichenarrays immer ein Element mehr zu veranschlagen, als für den eigentlichen String notwendig wäre. Bei allen anderen Arrays existieren keine solchen Vorschriften, es werden also nicht mehr Einträge definiert, als für die Daten benötigt werden. Dabei ist eine Wertzuweisung eines Elementes nur unter der Angabe des Indices möglich, z.B.:

```
werte[0] = 4711;
werte[1] = 707;
werte[2] = 31415;
```

Wie Sie sehen, wird beim Zählen der Indices stets mit 0 begonnen. Mit der gleichen Methode könnte man auch Texte an Strings zuweisen.

```
char string[80];

string[0] = 'O';
string[1] = 'K';
string[2] = '\0';
```

Unter uns, durch Verwendung der strcpy-Funktion kann man sich den ganzen Aufwand ersparen. Obige Zuweisungsfolge würde den String "OK" in der Variablen "string" ablegen, der mit dem üblichen Endekennzeichen '\0' abgeschlossen wird.

An dieser Stelle möchte ich nochmals den Unterschied zwischen Zeichen und Zeichenkette deutlich machen. Gerade der Anfänger, der vielleicht von einer anderen Sprache an C herantritt, kann durch seine Gewohnheit diese wichtigen Zusammenhänge übersehen. Der Unterschied zwischen "K" und 'K' liegt darin begründet, daß es sich bei "K" um einen String, bei 'K' um ein Zeichen handelt. Wird ein Buchstabe, ebenso wie Zeichenketten, in "Gänsefüßchen" eingeschlossen, so handelt es sich um einen String. Dieser ist aber mit einem '\0' Zeichen abgeschlossen, so daß "K" aus zwei Zeichen, nämlich 'K' und '\0' besteht. 'K' ist jedoch lediglich ein Zeichen. Dieser Umstand muß stets beachtet werden, da auch alle Routinen des Betriebssytems, die Sie vielleicht benutzen wollen, davon ausgehen, daß die Zeichenkette mit '\0' abgeschlossen ist. Der letzte Eintrag, auf den Sie zugreifen dürfen, ist nach der obigen Definition von "string[80]" über den Index 79 erreichbar (Man fängt ja beim Zählen mit 0 an).

13.1 Mehrdimensionale Felder

Die eben benutzten Arrays waren stets eindimensionale Felder, also Variablen, die nur einen Index verwenden. Mehrdimensionale Felder, z.B. zum Speichern der Stellung einer Schachpartie mit 8 * 8 Feldern, lassen sich mit der Definition von

```
int feld[8][8];
```

bewerkstelligen. Beide Indices liegen selbstverständlich zwischen 0 und 7. Um auf ein einzelnes Feld zugreifen zu können, sind nun zwei Indices nötig:

```
printf("Inhalt von Reihe 2 Spalte 4 %d\n",feld[1][3]);
```

Ebenso sind Definitionen zugelassen, die zum Beispiel fünf Indices fordern:

```
long inhalt[4][5][6][7][8];
```

Dabei ist aber zu beachten, daß Definitionen in dieser Weise sehr schnell gigantische Größenordnungen annehmen können, auch wenn es nicht so aussieht. Obiges Array würde 4 * 5 * 6 * 7 * 8 * 4 (Größe eines einzelnen long-Element) gleich 26880 Bytes also 26,25 KByte belegen.

Im Speicher des Rechners können die Daten aber nicht anders als unmittelbar hintereinander abgelegt werden. Von der Vorstellung, daß zweidimensionale Arrays beispielsweise tabellarisch nebeneinander liegen, müssen Sie abkommen. Wie würde denn sonst ein fünfdimensionales Array untergebracht? Da also alle Elemente in einer langen Folge (eindimensional) gespeichert werden, gibt es Regeln, wie dies zu erfolgen hat. Der erste Index wechselt erst dann, wenn alle Elemente, die zu seiner Gruppe gehören, untergebracht sind. Beim zweiten Index passiert dies schon etwas häufiger, und der letzte Index ändert sich bei jedem folgenden Element. Besser ist dieser Zustand in einer Liste zu erfassen, die die Lage der Einträge im Speicher wiedergibt. Vorausgesetzt wurde die Definition "int pos[4][3];":

Einträge innerhalb des Speichers
[0][0]
[0][1]
[0][2]
[1][0]
[1][1]
[1][2]
[2][0]
...
[3][1]
[3][2]

Zum Abschluß dieses Kapitels möchte ich Ihnen ein Programm vorstellen, das nicht nur mit Arrays operiert, sondern auch viele zuvor abgehandelten Themen noch einmal anschneidet. Das Programm fragt eine Reihe von Zahlen ab, übergibt diese dann einer Addierroutine und erhält die Summe zurück. Anschließend macht es noch einige statistische Auswertungen, damit sich das Abspeichern aller Werte überhaupt lohnt. Zum Ablegen der Eingabedaten wird natürlich ein Array verwendet, sonst wäre es an dieser Stelle fehl am Platze. Aber es sind auch einige andere Tricks darin untergebracht, die Sie sich genauer ansehen sollten.

```
#define FALSE 0
#define TRUE 1
#define MAXEINTRAG 20

long addiere(); /* Deklaration der Funktion */


main()
{
  int i, anzahl, ende = FALSE;
  long summe, eintrag[MAXEINTRAG];

  for(i = 0; i < MAXEINTRAG && !ende; i++)
    {
       printf("Bitte den %d. Wert: ", i+1);
       scanf("%6ld", &eintrag[i]);  /* hoechstens 6 Stellen */
       if(!eintrag[i])
         ende = TRUE;
    }
```

```
  anzahl = i - ende; /* Wenn letzter Wert 0, dann eins weniger */
  summe = addiere(eintrag, anzahl);
  printf("Die Summe aller %d Werte ist %ld\n", anzahl, summe);
    printf("Abweichung vom Mittelwert %.9lf:\n", (double) summe /
anzahl);
  for(i=0; eintrag[i] > 0; i++)
    printf("Wert %d: %-5.9lf%%\n",i+1,
      eintrag[i] * 100.0 / ( (double) summe / anzahl ) - 100.0);

}


long addiere(array, anz)
long array[];
int anz;
{
  long summe = 0;

  while(anz--)  /* knapp und praezise! */
    summe += array[anz];
  return summe;
}
```

Zuerst noch einige Informationen zum Programm. Um es etwas abzusichern, werden maximal 20 Eingaben zugelassen. Durch Verwendung eines Defines "MAXEINTRAG" läßt sich das Programm schnell an andere Größenvorstellungen anpassen. Wichtig ist im folgenden die Deklaration der Funktion "addiere". Da wir von dieser Funktion long-Werte verlangen, muß es auch dem Compiler mitgeteilt werden. Die Deklaration könnten wir ebensogut innerhalb der main-Funktion vornehmen, dort, wo auch alle anderen Variablen aufgelistet werden.

In der Abbruchbedingung der for-Schleife benutzen wir zur Abwechslung einmal den &&-Operator, der - Sie werden sich sicher noch erinnern - zwei Aussagen logisch durch UND verknüpft. Solange noch nicht alle Einträge belegt wurden und die Variable "ende" nicht 0 ist, wird die Schleife durchlaufen. Durch den Negationsoperator wird "ende" in das logische Gegenteil verkehrt. Zu Beginn enthält diese Variable den Wert 0, so daß der Ausdruck "!ende" 1 wird. Umgekehrt verhält sich die Sache, wenn "ende" auf 1 gesetzt ist und die Negation "!ende" zum

Verlassen der Schleife dient. Dies geschieht, wenn der Benutzer die Zahl 0 eingibt, die zum Beenden der Eingabe vorgesehen ist.

Der erste Eintrag im Array benötigt den Index 0. Da wir aber in der Regel bei 1 zu zählen anfangen, addieren wir bei der Textausgabe zum aktuellen Index eins hinzu. Jetzt zur scanf-Funktion, die wir schon so oft benutzt haben, daß wir sie eigentlich in- und auswendig kennen sollten. Die Formulierung

```
&eintrag[i]
```

ist das Wichtigste. Wenn wir ein anderes Array (bisher ausnahmslos Strings) eingegeben haben, tauchte kein "&"-Zeichen auf. Gerade das war scheinbar die große Ausnahme. Weil wir aber "eintrag[i]" und nicht "eintrag" dort hingeschrieben haben, handelt es sich ja überhaupt nicht um ein Array, sondern um eine ganz normale long-Variable. Und die wird wie alle elementaren Datentypen bei der scanf-Routine mit dem "&" ausgerüstet. Daß sich diese Variable innerhalb einer langen Kette gleichartiger Elemente, also einem Array, befindet, ist der scanf-Funktion egal, da sich daraus keinerlei Beschränkungen oder Sonderbehandlungen ergeben.

Die folgende if-Abfrage verdient ebenfalls Beachtung. Hier finden wir einen typischen Fall von C-Abkürzung. Die Abfrage soll ja in die Variable "ende" den Wert Eins übertragen, wenn die gerade getätigte Eingabe eine 0 war. Ausführlich würde dort dies vorzufinden sein:

```
if(eintrag[i] == 0)
  ...
```

Wenn "eintrag[i]" eine Null enthält, ist dieser Ausdruck auch Null. Mit Hilfe des Negationsoperators erhalten wir wieder eine wahre Aussage. Besonders bei Abfragen auf "== 0" oder "!= 0", treten die Abkürzungen oft an die Stelle, an der man normalerweise einen expliziten Wert erwartet. Komplizierter wird die Angelegenheit dadurch nicht, man muß nur wissen, was sich dahinter verbirgt.

Fahren wir bei der Analyse des Programms hinter der Schleife fort. Wurde die Schleife beendet, sei es, weil 20 Einträge gefüllt, oder der letzte Wert 0 war, so ermitteln wir die Gesamtzahl der abgespeicherten Daten. Der Addierfunktion übergeben wir die nötigen Daten, nämlich das Array mit den Einträgen und die Anzahl zu addierender Werte. Von dieser Routine erhalten wir die Summe als Rückgabewert. Mit dieser Information läßt sich dann auch die Abweichung jedes einzelnen Eintrages vom Mittelwert berechnen. Wäre die einzige Aufgabe des Programms, eine Folge von Zahlen zu addieren, bräuchte man gar keine Arrays, sondern würde alle Werte direkt nach der Eingabe aufsummieren.

14. Schleifen

14.1 Feinheiten der for-Schleife

Was soll denn die for-Schleife hier? Die haben wir doch längst abgehandelt! So oder ähnlich dürften Sie wohl jetzt reagieren. Der Grund, warum uns die for-Schleife erneut "beehrt", ist ihre Flexibilität. Die Bestandteile der for-Konstruktion, Initialisierung, Bedingungstest und Weiterschalten haben wir bereits behandelt, die Beschränkung ist aber gar nicht nötig. Die einzelnen Bestandteile der for-Schleife sind durch Semikolons zu trennen. Es können aber mehrere Befehle innerhalb der Initialisierung und der Fortschaltung untergebracht werden. Die sind dann nicht, wie bei C sonst üblich, mit Semikolon getrennt, sondern benutzen zu diesem Zweck das Komma. Und so kann die for-Schleife dann benutzt werden:

```
for(summe = 0, i = 1; i <= 20; i++)
  summe += i;
```

Oder auch

```
for(i = 1, j = 0; i < 10; i += 2, j+=3)
  ...
```

Dies ist der übliche Aufbau einer for-Schleife. Da aber C einiges mit sich machen läßt, sei dieses Beispiel vorgestellt:

```
for(printf("Jetzt geht es los!"); ; printf("Bong\n"), i++)
  if( (c = eingabe()) == 'e')
    break;
```

Hier wird bei Beginn der Schleife der Text "Jetzt geht es los!" ausgegeben, dann geprüft, ob die Bedingung, die zwischen den Semikolons steht, wahr ist. Dies ist stets der Fall, da dort nichts eingetragen ist. Wenn Sie sich erinnern, ein Null-Wert stellt bei jeder Bedingung falsch dar, alles andere ist logisch wahr. Die Bedingung ist in obiger Schleife immer erfüllt, wodurch es praktisch eine Endlosschleife ist, die nur dann verlassen werden kann, wenn eine Eingabe von 'e' erfolgt ist.

14.2 break

Sollte die Abfrage bei if wahr sein, so kommt der break-Befehl zur Ausführung. "break" beendet jede Schleife sofort und veranlaßt das Programm, mit dem nächsten Befehl hinter der gerade durchlaufenen Schleife fortzufahren. Das Entkommen aus dieser Schleife ist also nur mit dem break-Befehl möglich.

Bei der Fortschaltung wird genauso unorthodox vorgegangen. Man findet dort einen printf-Aufruf, der nach jedem Durchlauf aufgerufen wird. Von der ursprünglichen Konstruktion ist also nicht viel übrig geblieben. Eine endlose for-Schleife, die weder eine Initialisierung noch eine Prüfung oder Fortschaltung durchführt, ist

```
for(;;)
  {
    ...
  }
```

Eine for-Schleife kann jederzeit durch eine while-Schleife ersetzt werden (und umgekehrt). Die allgemeine Form lautet:

```
for(ausdruck1;ausdruck2;ausdruck3)
  {
    weitere Befehle
  }
```

oder

```
ausdruck1;
while(ausdruck2)
  {
    weitere Befehle
    ausdruck3;
  }
```

14.3 continue

Continue stellt das Gegenstück zum break-Statement dar. Anstatt die Schleife sofort zu verlassen, wird die Schleife wieder an der Stelle angesprungen, die nach Abarbeitung des letzten Befehls innerhalb der Schleife wieder an der Reihe wäre. Bei den drei Schleifentypen wären dies:

1. der Schleifenrumpf der while-Schleife (innerhalb der Klammern)
2. die Fortschaltung der for-Schleife, also for(...; ...; WEITER)
3. der Befehl nach do, bei do-while

Beispiel:

```
berechnung(feld)
double feld[];
{
  int i;

  for(i=0; i<ANZAHL; i=i+1)
    {
      if(feld[i] == 0.0)
        continue;

      hier geht's weiter!
    }
}
```

Sollte ein Eintrag innerhalb von "feld" den Wert Null besitzen, so wird die continue-Anweisung ausgeführt. Das Programm wird an der Stelle "i=i+1" fortgeführt, als wäre der Schleifenblock beendet worden.

14.4 Die switch-Anweisung

Mit diesem Befehl können mehrere gleichartige Vergleiche komfortabel gehandhabt werden. Am besten zeigt dies das folgende kurze Programm:

```
main()
{
  int ziffer;

  printf("Bitte Zahl eingeben!\n");
  while(1)
    {
      scanf("%d", &ziffer);
      switch(ziffer)
        {
          case 9:
            printf("Größer als 8\n");
          case 8:
            printf("Größer als 7\n");
          case 7:
            printf("Größer als 6\n");
          case 6:
            printf("Größer als 5\n");
          case 5:
            printf("Größer als 4\n");
          case 4:
            printf("Größer als 3\n");
          case 3:
            printf("Größer als 2\n");
          case 2:
            printf("Größer als 1\n");
          case 1:
            printf("Größer als 0\n");
          case 0:
            printf("Ziffer!\n");
            break;
          default:
            printf("Keine einzelne Ziffer!\n");
        }
      if(ziffer == 4711)
        break;  /* Endlosschleife verlassen */
    }
}
```

Der switch-Anweisung wird der zu prüfende Wert mitgegeben (switch(c)). Innerhalb des Anweisungsblocks wird dieser Wert mit den hinter dem Schlüsselwort "case" stehenden Werten verglichen. Diesem Wert muß noch ein Doppelpunkt folgen, dahinter stehen dann die auszuführenden Anweisungen. Ist der Vergleich positiv, stimmen also beide Werte überein, so werden die Befehle nach der "case"-Anweisung ausgeführt. Wenn der Vergleich negativ war, wird der darauffolgende Vergleich getestet, also alle folgenden Befehle bis zum nächsten "case" übersprungen.

Mit dem C-Wort "default" können noch Befehle ausgeführt werden, wenn keiner der Vergleiche erfolgreich war. Wenn man diese Struktur mit einer if-Abfrage vergleichen möchte, so entspricht "default" dem "else"-Zweig der if-Konstruktion. Ist erst einmal eine Prüfung positiv verlaufen, werden alle folgenden Befehle ausgeführt. "Alle" ist genau so gemeint, wie es hier steht. Es wird auch nicht vor dem folgenden "case" Halt gemacht.

Sollte das Zeichen, welches zur Prüfung übergeben wird, z.B. eine 5 sein, dann werden sämtliche Anweisungen (auch die hinter case '4', '3' usw.) bis zum nächsten break-Befehl ausgeführt. Dieser veranlaßt nämlich den direkten Abbruch einer Schleife oder, wie in diesem Fall, der switch-Anweisung.

Auch wenn Sie mehrere Anweisungen hinter einem "case" aufführen wollen, benötigen Sie keine geschweiften Klammern, wie es sonst in C üblich ist. Bleibt noch anzumerken, daß Sie mit switch alle elementaren Datentypen außer Fließkommazahlen vergleichen können.

15. Zeiger und Adressen

In diesem Kapitel werden Sie mit dem wohl wichtigsten Bestandteil der Sprache C vertraut gemacht. Von den einen geliebt, von anderen verhaßt: die Zeiger. Wenn von Vor- und Nachteilen von C die Rede ist, wird hundertprozentig das Stichwort Zeiger oder (engl.) Pointer fallen. Mit ihrer Hilfe ist es möglich, sehr kurze und schnelle Routinen zu schreiben, andererseits stehen viele Programmierer, die noch nicht mit Zeigern gearbeitet haben, wie der Ochs' vorm Berg.

15.1 Adresse

Fangen wir aber vorne an und überstürzen nichts. Zu dem Begriff Zeiger gehört untrennbar verbunden der Begriff "Adresse". Damit haben wir, ob wir es wußten oder nicht, bereits intensiv gearbeitet haben. Beim Aufruf der scanf-Funktion nämlich, dort mußte vor die meisten Variablen der "&"-Operator gesetzt werden. In diesem Zusammenhang handelt es sich um den sogenannten Adreß-Operator. Mit dieser Konstruktion läßt sich die Speicheradresse einer Variablen ermitteln. Wie Sie wissen, werden alle Daten, seien es Fließkommazahlen, Integer oder einzelne Zeichen, irgendwo im Rechner abgelegt. Die Position, wo die Daten zu einer Variablen untergebracht sind, wird durch eine eindeutige Zahl festgelegt, die Adresse. Im allgemeinen wird die Adresse immer mit der Hausnummer in einer langen Straße verglichen. Genau diese Nummer erhalten wir von der Variablen, vor die wir das "&"-Zeichen setzen.

Nehmen wir an, die Variable "a" wäre definiert worden und befände sich ab Adresse 100 gespeichert, dann würde der Ausdruck &a den Wert 100 liefern. Wozu braucht man aber überhaupt die Adressen, wenn es auch ohne geht?

Wenn Sie schon ein wenig mit eigenen Funktionen experimentiert haben, könnten Sie vor dem Problem gestanden haben, daß die aufgerufene Funktion vielleicht mehr als den einen, von "return" zurückgelieferten Wert der aufrufenden Funktion übergeben soll. Ebenfalls ist es nicht ohne besondere Tricks möglich,

die Variableninhalte einer, in einer anderen Funktion definierten Variablen zu ändern. Sehen wir uns hierzu folgenden Ausschnitt aus einem Programm an:

```
{
  int zahl = 6;
  aendern(zahl);
  ...
}

aendern(neuzahl)
int neuzahl;
{
  neuzahl = 5;
}
```

Der Funktion "aendern" erhält beim Aufruf lediglich eine Kopie des Inhaltes von "zahl". Wird in dieser Funktion der Wert dieser Variablen (neuzahl) geändert, so bleibt das Original, das sich in der aufrufenden Funktion befindet, unberührt davon. Das haben wir für unsere Zwecke ja schon genutzt. Sehen Sie sich zum Beispiel das Programm an, das die Summe einzelner Array-Einträge berechnet. Die Variable "anz" wird bis auf Null dekrementiert, während die Variable "anzahl" später zur Berechnung des Mittelwertes herangezogen wird.

Hier existiert nun die Möglichkeit, die Adresse der Variablen zu übergeben, so daß die aufgerufene Funktion direkt darauf zugreifen kann und die Funktion nicht, wie es normalerweise der Fall ist, eine Kopie der Variablen erhält. In welchem Datentyp soll jetzt aber diese Adresse gespeichert werden?

15.2 Zeiger oder Pointer

Prinzipiell könnte die Adresse in einer int- oder long-Variablen abgelegt werden. Es hängt von der Größe des int-Typs (compilerabhängig) und dem Prozessor ab, wie viele Bits für eine Adresse benötigt werden. Beim AMIGA braucht man 32 Bit, also ein long-Wert. Diese Möglichkeit bieten aber nicht alle C-Compiler, und das ist auch gut so. C läßt zwar einiges mit sich machen, man sollte es im eigenen Interesse aber nicht übertrei-

ben. Das Speichern von Adressen in long-Variablen ist eine äußerst unsaubere Programmierung. Viel besser und vor allem sicherer ist es, den dafür bereitgestellten Datentyp, die Zeiger zu verwenden. Ein Zeiger wird durch ein spezielles Zeichen markiert, dem Pointer "*". Der Zeiger ist aber mehr als bloßer Ersatz für eine long-Variable, da auch der Datentyp bei der Definition eines Zeigers mitangegeben wird. Welche Funktion übernimmt denn nun der Pointer? Wie bereits gesagt, soll er eine Adresse aufnehmen. Mit dieser Adresse kann er auf das entsprechende Objekt, hier der Inhalt einer Variablen, zugreifen und es manipulieren. Bei der Definition erhält der Zeiger zusätzlich Hinweise, um welchen Datentyp es sich dabei handelt. Er weiß also, auf welche Werte er zeigt. Beispiel gefällig?

```
char text[80];
char *zeiger; /* Definition als Zeiger auf char-Elemente */

text[6] = 'a';
zeiger = &text[6];
```

Gehen wir eine Zeile nach der anderen durch. Der erste Befehl, die Definition eines char-Arrays (String), ist klar. Nun folgt die Definition des Zeigers, den ich "zeiger" getauft habe. Vor dem Namen des Zeigers steht der Stern "*", der ihn als solchen ausweist. Zusätzlich, wie bei allen anderen Variablendefinitionen auch, ist der Datentyp angegeben. In das Element mit dem Index 6 (7. Eintrag) wird in der folgenden Zeile das Zeichen 'a' gespeichert. Nun kommt der Auftritt des Zeigers, der sich mittels Adreß-Operator "&" die Adresse des Elements 6 aus dem char-Array besorgt. Dadurch, daß "zeiger" nun die Adresse dieses Elements enthält, zeigt er auf das Zeichen 'a'. Der Zeiger verweist also auf eine andere Variable, nämlich auf text[6]. Dieser Verweis wird auch Referenz genannt, so daß die Umkehrung dieses Prozesses als Dereferenzierung bezeichnet wird. Diese Begriffe sind zwar nicht gerade sehr einfach zu behalten, müssen aber mal genannt werden, da sie Ihnen bestimmt häufiger begegnen werden.

Nun kann aber über den initialisierten Zeiger auf den Buchstaben zugegriffen werden. Der nächste Befehl könnte lauten:

```
if(*zeiger == 'a')
  printf("Er ist es!\n");
```

Soll auf das Element, das in der Speicherstelle steht, zugegriffen werden, so muß vor die Zeigervariable der Stern gesetzt werden. Daher ist der Ausdruck "*zeiger" ein Synonym für "text[6]" (natürlich nur, wenn "zeiger" auf diese Position zeigt). Auch die Änderung des Inhaltes dieser Speicherstelle ist jetzt über den Zeiger möglich:

```
*zeiger = 'b';
```

Nach dieser Anweisung würde nicht mehr 'a', sondern 'b' in der Speicherstelle stehen, auf die der Zeiger weist. Ohne das Array zu benutzen, haben wir also dessen Inhalt verändern können.

15.2.1 Tauschfunktion mit Zeigern

Kommen wir zu einer Routine, die auf jeden Fall die Werte der aufrufenden Funktion ändern sollte, die Tausch-Funktion:

```
tauschen(xp,yp)
int *xp, *yp;
{
  int help = *xp;
  *xp = *yp;
  *yp = help;
}
```

Diese Funktion erwartet als Übergabeparameter zwei Zeiger auf int-Werte. Zum Tauschen wird der erste Wert, auf den "xp" zeigt, in die Integer-Variable "help" gerettet, danach werden die Werte vertauscht. Der Aufruf der Funktion muß aber im Vergleich zu dem bisher üblichen ebenfalls geändert werden, da ja keine int-Werte, sondern Zeiger darauf (deren Adresse) erwartet werden:

```
int wert1, wert2;
wert1 = 3;
wert2 = 5;

tauschen(&wert1,&wert2);
```

Vielleicht werden Sie jetzt auch verstehen, warum wir bei scanf jedesmal den Adreß-Operator benutzen mußten. Mit dieser Funktion werden ja Daten in von uns gelieferte Variablen geschrieben, und das geht halt nur mit Zeigern und Adressen.

Bei Arrays, insbesondere bei den wohl häufig vorkommenden Zeichenketten, ist auf das einzelne Element erst komplett mit dem Index zuzugreifen. Daher muß man sich, wie bereits oben gezeigt, bei einem einzelnen Eintrag die Adresse in der Form "&array[index]" besorgen. Für das erste Element in dieser Liste müßte man folgendes konstruieren:

```
&array[0]
```

In C stellt der Name eines Arrays aber nichts anderes als die Speicheradresse des ersten Elements dar. Daher kann mal wieder abgekürzt werden, für "&array[0]" schreibt man "array". Beide liefern die Adresse des ersten Elementes, nicht dessen Inhalt. Der Name eines Arrays ist praktisch schon ein Zeiger, der auf das erste Element verweist. Jetzt ist auch klar, warum beim scanf-Aufruf der Name des Strings nicht mit dem Adreß-Operator "&" versehen werden wußte, er stellt die Adresse bereits selbst dar:

```
char string[81];
scanf("%s",string);
```

Es war keine Ausnahme, wie wir vermuten mußten, sondern erneut eine Kurzversion von "&string[0]".

15.2.2 strcpy Version 2

Eine ideale Anwendung von Zeigern bietet die Funktion zum Kopieren von Strings. Durch Verwendung von Pointern können die Indices, die wir bei der ersten Formulierung von strcpy noch benutzen mußten, gespart werden. Ein Beispiel dafür zeigt die folgende Konstruktion mittels Zeiger.

```
strcpy(nach,von) /* Version 2 */
char *nach, *von;
{
  while((*nach = *von) != '\0')
    {
      nach++;
      von++;
    }
}
```

In der obigen Routine strcpy wird auch die Besonderheit der Zeiger deutlich: Erhöht man den Zeiger um Eins, so zeigt der Zeiger auf das folgende Element, erhöht man es um Zwei, so zeigt er auf das übernächste. In dieser strcpy-Version wird solange ein Zeichen, auf das "von" zeigt, in die Adresse, auf die "nach" zeigt, übertragen, bis der übertragene Wert gleich 0 ist. Dann hat nämlich der Ausdruck (*nach = *von) den Wert 0 und bewirkt in der Prüfung auf ungleich 0 den Abbruch der while-Schleife. Das letzte noch übertragene Zeichen ist dieses gerade getestete Nullbyte, das ja das Endekennzeichen eines Strings darstellt.

15.2.3 strcpy Version 3

Es wäre aber kein C-Programm, wenn es sich nicht noch kürzer formulieren ließe. Ein Test auf Null kann meistens irgendwie umgangen werden, und das Inkrementieren der Zeiger kann auch noch in die Abbruchbedingung gequetscht werden. Kürzer also

```
strcpy(nach,von) /* Version 3 */
char *nach, *von;
{
  while(*nach++ = *von++)
    ;
}
```

Das dürfte eine der kürzesten und schnellsten Versionen zum Kopieren von Strings sein, die nur noch durch einen besonderen Trick schneller gemacht werden kann. Dazu aber später.

Möchten Sie ein Programm schreiben, das nicht char-, sondern float-Werte aus einem in ein anderes Array überträgt, so ist bei obiger Formulierung nur ein einziges Wort zu ändern: char. An dessen Stelle setzen wir den Datentyp float und schon können float-Werte kopiert werden, die völlig anders aufgebaut und auch viel mehr Speicherplatz pro Element verbrauchen. Wie ist das eigentlich möglich, daß das mit den Zeigern trotzdem so hinhaut?

Durch die Definition

```
float *nach, *von;
```

wird dem Programm mitgeteilt, daß die Zeiger "von" und "nach" auf Werte des Datentyps float verweisen. Dieser Datentyp verbraucht in der Regel pro Eintrag 4 Bytes. Wird nun ein so definierter Zeiger um Eins erhöht, z.B. nach++, so zeigt er auf das folgende Element. Dieses liegt zwar nun 4 Bytes vom ursprünglichen Element entfernt, doch das weiß der Compiler durch die Definition des Zeigers. Mit dem Erhöhen oder Erniedrigen des Zeigers um 3 würde die Adresse sich in Wirklichkeit um 12 Bytes ändern. Beim Datentyp double, der normalerweise 8 Bytes verbraucht, wird auch das berücksichtigt. Pro Inkrementierung des Zeigers ändert sich die Adresse um 8 Bytes. Wie Sie sehen, ist so ein Zeiger eine lobenswerte Einrichtung!

Wie verarbeitet nun der Compiler aber Ausdrücke wie "string[4]", wenn diese Gruppen so verwandt miteinander sind? Da "string" der Name des Arrays ist, der in C der Adresse des ersten Eintrages (string[0]) entspricht, formt der Compiler diesen

Ausdruck erst noch in das Äquivalent *(string + 4) um. Zuerst wird auf die Adresse "string" die Länge von 4 Elementen addiert, so daß der jetzige Pointer auf den Eintrag string[4] zeigt, dann wird über den Stern auf dieses Element zugegriffen. Die Klammern sind nötig, weil der Pointer "*" eine höhere Priorität als die Addition besitzt (eine Tabelle dazu finden Sie im Anhang). Einen Vergleich zwischen Pointer- und Array-Schreibweise soll folgende kleine Gegenüberstellung verdeutlichen:

```
long wert, daten[10]; /* So wurde definiert */
```

Arrayschreibweise	Dasselbe mit Pointern
`wert = daten[3];`	`wert = *(daten + 3);`
`daten[0] = wert;`	`*daten = wert;`
`daten[7] += wert;`	`*(daten + 7) += wert;`

Wie bereits im obigen Programm gezeigt, kann in der Konstruktion "*zeiger" ebenfalls mit weiteren Operatoren gearbeitet werden. So bedeutet beispielsweise die verwendete "*nach++"-Anweisung bei strcpy, daß zuerst der Wert geholt werden soll, auf den "nach" zeigt (*nach) und danach der Zeiger auf das nächste Feld zeigen soll (++). Können Sie sich vorstellen, was folgende Befehle durchführen würden?

```
int i, *ip = &i;
i = 100;
--*ip;
```

Nach der Definition der int-Variablen "i" und des int-Pointers "ip", der hier gleichzeitig noch initialisiert wird, erhält auch die Variable "i" einen Wert zugeordnet. In ihr wird die Zahl 100 abgelegt. Und jetzt kommt die große Frage, was produziert "--*ip"?

Zuerst wird einmal die Zahl geholt, auf die ip zeigt (*ip), nämlich 100. Dann wird der gelieferte Wert um Eins verringert, aus 100 wird 99. Dieser Wert steht nun in der Variablen i. Das gleiche Resultat hätte man durch den wesentlich einfacheren

Ausdruck --i bekommen können, aber irgendwie müssen wir ja den Umgang mit Pointern lernen.

15.3 Zeiger ohne Speicher

Bei der Verwendung von Zeigern ist stets darauf zu achten, daß sie eben nur einen Zeiger auf einen bestimmten Datentyp darstellen. Der Speicherplatz für die einzelnen Elemente muß separat definiert werden und der Zeiger darauf gesetzt werden. Die Initialisierung des Zeigers ist nicht nur nötig, um unsinnige Ergebnisse zu vermeiden, sondern, was fast immer bei nicht initialisierten Zeigern der Fall ist, um einen Absturz inclusive Guru-Meditation-Service zu verhindern. Sollten Sie während eines Testlaufes Ihres Programmes bei Verwendung von Zeigern einen Absturz miterleben, so sollten Sie zuerst überprüfen, wohin der Pointer oder eventuell der Index eines Arrays (das kommt auf's gleiche raus) zeigt.

Gelegentlich finden Sie solche Programme, die anscheinend diesen Forderungen widersprechen:

```
main()
{
  char *text_ptr;

  text_ptr = "Alle zeigen immer auf mich!";
  printf("Der Text lautet >%s<\n",text_ptr);
}
```

Wo ist in diesem Programm denn der Speicherplatz für den String? Vom Pointer wird in diesem Zusammenhang nichts unternommen. Dieser wird irgendwo zwischen dem Programmtext abgelegt, genauso, wie Sie es bei Funktionsaufrufen wie "printf("Hallo\n");" erwarten. Auch dieser String innerhalb der Funktion muß irgendwo gespeichert werden.

Aber Achtung!

Sollten Sie den Text ändern wollen, beispielsweise durch Zugriff mittels text_ptr, so müssen Sie unbedingt die maximale Länge beachten. Bei obigen String beträgt diese nur 28 Zeichen, wobei ein Zeichen noch für das Stringende '\0' draufgeht. Schreiben Sie in den Bereich trotzdem 30 Zeichen hinein, so können Sie getrost einen satten Absturz erwarten. Es ist nämlich nicht auszuschließen, daß hinter dem String noch Programmcode steht, der dann ebenfalls überschrieben wird. Sollte der Prozessor auf solche, für ihn unverständliche Daten stoßen, so gerät er außer Rand und Band.

Wie Sie bereits erfahren haben, symbolisiert der Name eines Arrays das erste Element dieser Kette. Nun die Frage, was ist der Ausdruck feld[3][2], wenn die folgende Definition zugrunde legt?

```
int feld[5][5][10];
```

Ist es ein Element dieses Arrays, wenn ja, welches, wenn nein, was ist es dann? Tja, eine harte Nuß! Sehen Sie sich den zu untersuchenden Ausdruck genau an! Er enthält nur zwei Indices, bei der Definition sind jedoch drei angegeben. Daraus folgt, daß es schon mal kein Element sein kann. Es kann also nur ein Zeiger sein, der auf das erste (?) Element zeigt. Als erstes Element ist nicht feld[0][0][0], sondern das erste Feld, auf das feld[3][2] zeigt, gemeint: feld[3][2][0]. Ist Ihnen nun klar, welche wundersame Wandlung das Vergessen eines Index zur Folge hat? Aus einem Element des Feldes wird ein Zeiger auf ein Feld, bei dem die fehlenden Indices durch "[0]" ersetzt wurden. feld[3] zeigt somit auf feld[3][0][0]. Wenn also etwas möglich ist, dann läßt es sich mit Zeigern machen. Später werden wir noch einige akrobatische Kunststücke mit Zeigern bewundern können.

16. Speicherklassen

Unter diesem Begriff versteht man verschiedene Gruppen von Variablen, die eine unterschiedliche Lebensdauer während des Programmlaufes besitzen. Es existieren 4 Speicherklassen: auto oder lokal, global, register und static. Welche Funktionen haben jetzt diese Speicherklassen?

16.1 auto

Ohne uns groß darum zu kümmern, haben wir die ganze Zeit mit auto-Variablen oder auch lokalen Variablen gearbeitet. Diese Variablen gehören deshalb zur auto-Klasse, weil sie bei jedem Aufruf einer Funktion AUTOmatisch neu definiert werden und ihnen Speicherplatz zur Verfügung gestellt wird. Ihre Lebenszeit ist auf die Ausführungszeit der Funktion begrenzt. Nach dem Verlassen der Funktion durch return oder Erreichen der letzten geschweiften Klammer dieser Funktion wird der zuvor belegte Speicherplatz wieder freigegeben und kann für andere Aufgaben eingesetzt werden. Diese lokalen Variablen können also nur in der Funktion benutzt werden, in der sie auch definiert wurden. Der Inhalt der Variablen ist verloren, und der Name ist nun nirgendwo innerhalb des Programmes mehr bekannt. Dies waren die lokalen Variablen.

16.2 static

Im Gegensatz dazu bleiben die static-Variablen bis zum Programmende erhalten und werden nicht beim Verlassen der Funktion gelöscht, um bei einem erneuten Aufruf der Funktion wieder kreiert zu werden. Mit Verlassen ist hier stets die Beendigung der laufenden Funktion gemeint, was nicht mit einem weiteren Funktionsaufruf innerhalb dieser Funktion zu verwechseln ist. Die Kontrolle geht zwar kurzfristig auf eine andere Routine über, die aufrufende Routine ist aber immernoch aktiv (schließlich erwartet diese ja noch ein Ergebnis). Wenden wir uns hierzu einer Anwendung von static-Variablen zu:

Das C-Wort "static" wird einfach vor eine Definition gestellt, z.B.:

```
funktion()
{
  static int zaehler = 1;
  ...
}
```

Beim ersten Aufruf der Funktion wird die Variable definiert und, falls wie im obigen Beispiel gewünscht, mit einem Startwert initialisiert. Verläßt man die Funktion zwischenzeitlich, so wird beim erneuten Funktionsaufruf keine neue Variable angelegt, da sie immer noch existiert. Auch ihr Inhalt bleibt erhalten, so daß sie auch nicht noch einmal initialisiert wird. So kann man in einer Routine beispielsweise mitzählen, wie oft sie bereits aufgerufen wurde.

16.3 global

Als weitere Speicherklasse sind die externen oder globalen Variablen zu nennen. Diese Variablen werden außerhalb aller Funktionen definiert und können auch von allen Funktionen benutzt werden. Ein Ausschnitt aus einem Programm würde so aussehen:

```
#define EOS '\0'

int error, dummy;

main()
{ ...
}
```

Die Variablen, die so definiert wurden, können auch von Funktionen benutzt werden, die sich nicht in der Sourcedatei befinden. Wie Sie wissen, werden dem Linker eine Reihe von Dateien zum Linken übergeben. Diese Dateien enthalten bereits compilierte Funktionen die vielleicht globale Variablen benötigen, die

in Ihrem Programm erst noch die richtigen Werte zugewiesen bekommen müssen. Solche Variablen müssen vor Gebrauch nur mit dem C-Befehl "extern" deklariert, nicht definiert werden:

```
extern int error;
```

Dadurch kann diese Variable auch in einer Datei verwendet werden, in der "error" überhaupt nicht definiert worden ist.

Es sind auch Kombinationen, wie z.B. globale static-Variablen zugelassen. Durch diese Definition können zwar alle Funktionen an die globale Variable innerhalb der Quelldatei heran, aber die eben geschilderte Situation, daß aus einer anderen Datei heraus eine Funktion auf diese Variable zugreifen kann, wird verhindert. Die Variable ist nur in der Quelldatei bekannt. Funktionen, die erst beim Linker in Kontakt mit unserem Programm treten, haben zu dieser Variablen keinen Zugang.

16.4 register

Die letzte Speicherklasse ist "register". Wer schon ein bißchen in Assembler programmiert hat, wird sofort wissen, worum es sich handelt. Ein Prozessor, der wichtigste Teil eines Rechners (das Gehirn), besitzt verschiedene interne Speicher. Ein solcher Speicher, der nicht mit dem RAM des Computer verwechselt werden sollte, wird auch Register genannt. Die Anzahl der zu benutzenden Register ist natürlich stark von dem verwendeten Prozessor abhängig. So besitzt ein 6502/10, der im C64 oder den ATARI 600/800/130 eingebaut ist, nur 3 Register (2 Register und einen Akkumulator), während der im AMIGA, ATARI ST und Macintosh verwendete M68000 17 Register sein eigen nennt, die dazu noch 4mal so groß sind wie beim 6502. Daher wird kaum ein C-Compiler für 6502-Rechner die Möglichkeit anbieten, Register als Variablenspeicher zu verwenden. Wir haben aber wie gesagt 17 Register des Prozessors, von denen uns je nach Compiler 3 - 5 Register zur Verfügung gestellt werden. Die Restlichen werden für andere interne Aufgaben benötigt.

Eine als "register" definierte Variable muß also innerhalb eines Registers Platz finden. Beim M68000 sind dies 32 Bit oder 4 Byte, so daß als Datentypen nur Integerzahlen zugelassen sind. Auch wenn der float-Wert nur 4 Byte belegen sollte, kann er nicht in einem Register abgelegt werden. Das wurde so festgelegt, und alle halten sich daran! Gültige Datentypen wären:

> int, char, short, unsigned, long, Kombinationen davon und Zeiger

Zeiger sind deshalb möglich, da sie eigentlich nur die Adresse eines Objektes darstellen, und die belegt beim AMIGA bekanntlich 4 Bytes. Paßt genau!

Es gibt aber noch weitere Beschränkungen: So darf die so definierte Variable nur eine auto-Variable sein, da für sie ja ein Register der Zentraleinheit belegt wird. Diese sind rar und können nur kurzfristig zur Verfügung gestellt werden. Nach Verlassen der Funktion, in der sie definiert wurde, wird das Register wieder frei für andere Aufgaben.

Der Vorteil der Register-Variablen ist der enorme Geschwindigkeitsvorteil. Diesen kann das Programm aber nur dann voll ausschöpfen, wenn solche Variablen bei vielen Schleifendurchläufen oder Berechnungen verwendet werden. Die Variable muß nicht bei jeder Benutzung aus dem Speicher in ein Register geladen werden, sondern befindet sich bereits dort.

16.4.1 Schnelle strcpy-Routine

Bevor wir das erste Beispiel präsentieren noch eine Einschränkung. Es ist nicht möglich die Adresse einer register-Variablen mittels "&"-Operator zu erhalten, da ein Register eben keine Adresse besitzt. Es befindet sich ja nicht im RAM.

```
strcpy(nach, von) /* letzte Version */
register char *nach, *von;
{
  while(*nach++ = *von++)
  ;
}
```

Durch diese Definition der char-Pointer als register dürfte wohl das Optimum an Geschwindigkeit in C herausgeholt sein. Noch schneller geht es dann nur, wenn Sie auf Maschinensprache umsteigen.

Wenn Sie wissen möchten, welchen Zeitvorteil Sie durch die Verwendung von Registern erreichen können, testen Sie Ihren Compiler doch mit folgendem Programm. Um die Geschwindigkeit einigermaßen messen zu können, muß das Programm möglichst oft die Register benutzen und sollte zum anderen keine anderen Funktionen benutzen, die nur die Zeitspanne verlängern. Deshalb hat das folgende Programm nichts anderes im Sinn, als eine Variable von 5000000 auf 0 herunterzuzählen. Etwas anderes kommt in der Schleife nicht in Frage.

```
#include <stdio.h>

main()
{
  printf("Zeitvergleich Mit und Ohne Register\nRETURN für Start\n");
  getchar();
  printf("%cStart Ohne", 7);
  ohne_register();
  printf("%cStop!\nRegisterroutine mit RETURN\n", 7);
  getchar();
  printf("%cStart Mit", 7);
  mit_register();
  printf("%cStop!\n\n", 7);
}
```

```
mit_register()
{
  register long i = 5000000;  /* Von 5000000 bis 0 zaehlen! */
  while(i--)
    ;
}


ohne_register()
{
  long i = 5000000;
  while(i--)
    ;
}
```

In diesem Programm nutzen wir nun auch zum ersten Mal den Präprozessor-Befehl "#include", mit dessen Hilfe die schon untersuchte "stdio.h"-Datei mit in unser C-Programm aufgenommen wird. Wir benötigen dieses File, weil wir eine neue Funktion, die getchar-Routine verwenden. Sie sollte ein Zeichen von der Tastatur liefern, so ist es zumindest üblich. Beim Lattice-C-Compiler wartet die Funktion auf die RETURN-Taste nach jedem Zeichen und widerspricht dadurch nicht nur jedwedem Standard, sondern ist dadurch fast vollkommen unbrauchbar. Zu einem Zweck läßt sie sich aber noch benutzen, zum Warten auf die RETURN-Taste, und das genügt uns ja im obigen Programm.

Handgestoppt komme ich auf Zeiten von 51.6 Sekunden ohne und 24.1 Sekunden mit Registervariablen. Das kann sich doch sehen lassen, mehr als zweimal schneller einzig durch Verwendung des Wörtchens "register". Dabei sollten Sie aber beachten, daß Sie nicht die Möglichkeiten des Multitasking beim AMIGA ausschöpfen und eventuell nebenbei noch ein Programm compilieren lassen. Dann erhalten Sie natürlich andere Ergebnisse.

16.5 Lokal

Kehren wir zurück zum eigentlichen Thema, den Speicherklassen und den lokalen Variablen. Lokale Variablen sind keine besondere Speicherklasse, sondern nur das Gegenteil von "global". Man kann verschiedene Variablengruppen wie register, auto (also ohne alles), static "lokal" definieren. Sie gelten immer nur in dem Block oder der Funktion, in der sie auch kreiert wurden. Dabei haben alle lokalen Vorrang vor globalen Variablen, d.h. sollten zwei Variablen mit dem gleichen Namen definiert werden (natürlich nicht im gleichen Befehlsblock), so wird die lokale Variable verwendet. Sie erhält den Vorzug, während die andere (globale Variable) für den Augenblick verschwindet und unsichtbar wird. Sehen wir uns dazu ein Beispiel an.

```
int i = 1;

main()
{
  int i = 2;
  printf("%3d", i);
    {
      printf("%3d", i);
        {
          int i = 3;
          printf("%3d", i);
        }
      printf("%3d", i);
    }
  printf("%3d", i);
  test();
  printf("\n\n");
}


test()
{
  printf("%3d", i);
    {
      int i = 4;
      printf("%3d", i);
    }
}
```

Es werden nacheinander die Ziffern 2, 2, 3, 2, 2, 1 und 4 ausgegeben. In der main-Funktion wird nämlich eine neue (lokale) Variable deklariert, so daß die globale Variable "i" nicht mehr ansprechbar ist. Im folgenden Block bleibt diese Konfiguration bestehen, somit erscheint erneut eine 2. Darauf folgt wieder ein Block, in dem diesmal aber eine weitere "i"-Variable definiert wird. Deshalb wird die zuvor benutzte für das Programm unsichtbar, und die eben definierte Variable tritt dafür hervor. Das Resultat der Ausgabe ist 3. Wenn danach alle Blöcke wieder verlassen werden, tauchen auch die "versteckten" Variablen wieder auf, da die zuvor ausgegebene Variable mit dem Wert 3 durch Verlassen des Blocks gelöscht wird und verschwindet. Nun wird die test-Funktion aufgerufen, die ihrerseits eine Ausgabe von "i" vornimmt. Da noch keine lokale Variable an dieser Stelle bekannt ist, wird zur Ausgabe die globale Variable benutzt: 1. Zuletzt wird auch hier eine lokale Integervariable ins Leben gerufen, die wiederum die globale überdeckt. Dadurch ist sichergestellt, daß immer die zuletzt in einem Block definierte Variable benutzt wird und oft benötigte Namen wie z.B. "i" oder "j" für Laufvariablen in vielen Schleifen als unterschiedliche Variablen erkannt werden.

17. Unsere Bibliothek

Ein Vorteil der C-Programmierung liegt unter anderem in dem modularen Aufbau von größeren C-Programmen. Diese benötigen oftmals viele Funktionen, die schon einmal bei früheren Produktionen verwendet wurden und bereits existieren. Mit dem schon erwähnten "#include"-Befehl ist es möglich, kleinere Dateien, die häufig benutzte Funktionen oder Makros (werden noch ausführlich behandelt) enthalten, in das aktuelle Programm einzubinden. Dies geschieht vor der Compilierung, so daß für den Compiler nur eine große Datei, nicht mehrere kleine vorhanden sind. Jeder C-Programmierer wird im Laufe seiner Tätigkeit damit anfangen, bestimmte Funktionen selbst zu schreiben und nach dem Test auf fehlerfreies Funktionieren in eine separate Datei abzuspeichern. Auch wir wollen nun damit beginnen, denn wir haben ja bereits zwei Funktionen: strcpy und strlen. Diese Basisfunktionen brauchen zwar nicht neu definiert zu werden, da alle Compiler diese Funktionen bereits in ihren Bibliotheken aufbewahren, zum besseren Verständnis sind sie aber sehr nützlich.

Schreiben Sie also die beiden Funktionen (und nur die beiden, keine main-Funktion) in eine eigene Datei mit dem Namen "string.c". In anderen Programmen können Sie sich diese Funktionen durch den Präprozessor-Befehl

```
#include "string.c"
```

oder

```
#include <string.c>
```

in Ihre aktuelle Datei einbinden. Zwischen den obigen Versionen besteht ein Unterschied, die erste Formulierung erwartet diese Datei im Hauptdirectory, die zweite ermöglicht es, die Dateien auch in Subdirectories unterzubringen und von dort in das Programm zu integrieren. Die zweite Version ist also flexibler.

Im Lieferumfang Ihres C-Compilers dürften sich eine ganze Reihe von Bindings, also Dateien zum Einbinden befinden. Diese mit ".h" gekennzeichneten Dateien enthalten hauptsächlich "#define"-Anweisungen für fast alle möglichen Dinge. Komplette Funktionen sind dort nicht enthalten, diese wurden bereits in eine Library z.B. "amiga.lib" oder "lc.lib" transportiert. Auch die Bindings (".h"-Dateien) können und sollen bei Bedarf eingebunden werden. Der entsprechende Befehl wäre

```
#include <datei.h>
```

Bevor wir jedoch mit dem Einbinden beginnen, wollen wir erst mal etwas schreiben, was wir einbinden können. Eine solche recht nützliche Funktion soll zum Vergleichen von Strings geschrieben werden. Da Strings keine elementaren Datentypes sind, kann man sie nicht durch

```
if(string1 == string2) /* Oh, wie falsch!!! */
  ...
```

vergleichen. Können Sie mir sagen, was wir da gerade vergleichen? Sollten wir die Variablen "string1" und "string2" als Zeichen-Array definiert haben, so entspricht der Name, wie wir schon wissen, der Adresse des ersten Elementes (&string1[0]). Wir vergleichen dadurch die Adressen der zwei Arrays und die sind immer verschieden. Da beide Arrays vom Compiler jeweils einen separaten Speicherplatz für ihre char-Einträge zugeordnet bekommen, ist die Abfrage völlig nutzlos. Der einzige (theoretisch) mögliche Fall, daß diese if-Abfrage erfüllt ist, tritt dann ein, wenn wir eine (oder beide) der Variablen als Zeiger definieren und auch auf den gleichen String zeigen lassen. So geht es also nicht.

17.1 strcmp

Vielmehr müssen wir uns selbst an die Arbeit machen und Schritt für Schritt jedes einzelne Element des ersten Strings mit denen des zweiten Strings vergleichen.

```
strcmp(s,t)
register char *s, *t;
{
  while(*s == *t)
  {
  if(!*s)
  return(0); /* Ende erreicht (*s == 0) */
  s++;
  t++;
  }
  return(*s - *t);
}
```

Die Funktion strcmp vergleicht die Zeichen des Strings "s" mit denen von "t". Solange die Zeichen gleich sind (*s == *t), wird die while-Schleife durchlaufen. Dabei wird überprüft, ob vielleicht gerade das letzte Zeichen, der '\0'-Marker EOS (End Of String) verglichen wurde. Ist dies der Fall, müssen beide Strings identisch sein, da hier "s" und "t" beendet sind. Andernfalls werden die Zeiger auf das nächste Element gesetzt, und der Vorgang wird wiederholt.

Taucht ein Zeichen innerhalb von "s" auf, welches von dem "t"-Zeichen verschieden ist, so wird die while-Schleife abgebrochen und die Differenz der beiden Zeichen (*s - *t) an das aufrufende Programm zurückgeliefert. Negative Werte zeigen an, daß der String "s" "kleiner" als "t" war, positive Werte symbolisieren das Gegenteil. Eine Null, die nach der if-Abfrage geliefert wird, weist daraufhin, daß beide Zeichenketten vollkommen identisch sind.

Packen Sie diese Funktion zu den beiden anderen Dateien in "stringfunk.c". Da wir mit unserem jetzigen Kenntnisstand auch die schon etwas ältere strlen-Routine noch verbessern können, benutzen wir dieses Mal die Zeiger. Anstatt des Indices wird der Zeiger über alle Einträge des Strings bis zum EOS-Zeichen

geführt. Zuvor müssen wir uns natürlich den Startwert merken, damit wir die Anzahl der Inkrementierungen errechnen können. Das ist schneller, als wenn wir jedesmal mit einer zusätzlichen Variablen mitzählen. Die Datei "stringfunk.c" sieht deshalb so aus:

```
/* Einige selbstdefinierte Stringfunktionen */


strcpy(nach, von)
register char *von, *nach;
{
  while(*nach++ = *von++)
    ;
}


strlen(s)    /* Umstellung auf Zeiger! */
register char *s;
{
  register char *help = s;  /* Anfangsposition sichern */
  while(*s)
    s++;
  return(s - help);  /* Differenz zwischen Zeigern ergibt
Elementzahl */
}


strcmp(s,t)
register char *s, *t;
{
  while(*s == *t)
    {
      if(!*s)
        return(0);  /* Ende erreicht (*s == 0) */
      s++;
      t++;
    }
  return(*s - *t);  /* Differenz zwischen den beiden ungleichen
Zeichen */
}
```

Nun wollen wir testen, ob sie auch richtig funktioniert. Dazu benutzen wir zwei Strings, die wir bereits im Programm initialisieren.

```
#include "stringfunk.c"

/* Globale Arrays konnen initialisiert werden! */

char string1[] = "Hallo!";
char string2[7] = { 'H', 'a', 'l', 'l', 'o', '!', 0};

main()
{
  printf("\nVergleich von >%s< und >%s< ist %d\n",
    string1, string2, strcmp(string1, string2));
  printf("Nun >%s< und >%s< Ergebnis %d\n\n",
    string1, "Huhu!", strcmp(string1, "Huhu!"));
}
```

Erst zu den vertrauten, erwarteten Ergebnissen des Funktionsaufrufes strcmp. Der erste Aufruf liefert Null, da beide Zeichenketten auch wirklich gleich sind, der zweite Aufruf liefert -20. Diese Zahl resultiert aus dem Vergleich der Zeichen 'a' und 'u', d.h. der erste String ("Hallo!") ist kleiner als der zweite ("Huhu").

Neu in diesem Programm ist die Initialisierung von Arrays. Bisher wurde jedes Element fein säuberlich einzeln belegt. Dies ist bei automatischen Variablen auch kaum anders machbar, bei globalen Variablen hingegen kann wie oben praktiziert, direkt ein String, oder, wie im zweiten Beispiel jedes Feld separat, initialisiert werden. Im ersten Beispiel wird noch nicht einmal angegeben, wie viele Elemente "string1[]" eigentlich haben soll. Wieder ein Indiz dafür, daß die Sprache C wohl genau für die erfunden wurde, für die das Motto gilt: Faulheit ist eine Tugend. Die Anzahl der Felder hat sich der Compiler gefälligst selbst herauszusuchen, wofür ist er denn sonst da. Er initialisiert das Feld "string1" also mit 7 Elementen (Nullbyte am Ende nicht vergessen!). Wer unbedingt möchte, kann, wie im zweiten Beispiel, diesen Wert auch angeben, aber wer macht sich schon gerne unnütze Arbeit?

Sollen die Elemente einzeln (Beispiel 2) den Feldern zugewiesen werden, so sind sie in geschweifte Klammern zu fassen und durch Kommata zu trennen. Bei mehreren Dimensionen sollten entsprechend viele geschweifte Klammern verwendet werden.

```
int feld[4][4]=
{
  { 1, 2, 3, 4 },
  { 6, 3, 4, 9 },
  { 3, 4, 5, 6 },
  { 12,9, 0, 2 },
};
```

Diese Formulierung weist dem feld[4][4] die entsprechenden Werte zu, dabei werden die ersten Werte "{ 1, 2, 3, 4 }" in den Feldern feld[0][0] bis feld[0][3] untergebracht. Die inneren Klammern sind i.d.R. nicht vorgeschrieben (beim Lattice allerdings schon!), die obige Anweisung könnte auch so aussehen:

```
int feld[4][4]= { 1, 2, 3, 4, 6, 3, 4, 9, 3, 4, 5, 6, 12, 9, 0, 2 };
```

Nicht gerade sehr übersichtlich, oder?

Wenn einige Elemente nicht initialisiert werden sollen, dann brauchen sie auch nicht angegeben zu werden. Alle ausgelassenen Elemente erhalten automatisch den Wert Null.

```
int feld[3][3]= {
                  { 3, 2 },
                  { 4 },
                  { 3, 4, 5 },
                };
```

Die Felder feld[0][[2], feld[1][[1] und feld[1][2] enthalten danach Null. Nach der Definition muß unbedingt ein Semikolon folgen, sonst nimmt es Ihnen der Compiler übel. Nach den inneren geschweiften Klammern muß ein Komma stehen, auch hinter der letzten. Und bedenken Sie bitte, diese Initialisierungen sind nicht bei auto-Variablen zugelassen, sondern nur auf globale oder static anzuwenden.

17.2 itoa

Eine weitere, sehr häufig anzutreffende Routine beim Umgang mit Strings, ist itoa (Integer TO Ascii). Sie wandelt einen Integerwert in die entsprechende Zeichenkette um, d.h. man übergibt die Zahl 123 und erhält in einem Zeichenarray "123" zurück. Dies ist besonders dann wichtig, wenn Texte komplett mit Zahlen aufbereitet werden sollen. Alle Umwandlungen, die uns sonst printf abnimmt, können wir auch mit eigenen Routinen erreichen. Wenden wir uns nun wieder der itoa-Funktion zu. Als Parameter braucht die Funktion einen Integerwert, den sie umwandeln kann, und einen String, in dem das Ergebnis abgelegt wird. Der Kopf der Funktionsdefinition lautet demnach:

```
itoa(n, s)
char s[];
int n;
```

Die Umwandlung wird durch den Modulo-Operator "%" durchgeführt. Dividieren wir immer die Zahl durch 10, so erhalten wir die letzte Stelle davon. Dazu wird noch der Code für die Ziffer '0' addiert, und schon ist das erste Zeichen erfaßt. Unsere Zahl wird dann durch 10 dividiert, so daß sie um eine Stelle nach links rückt und die gerade gewonnene letzte Ziffer wegfällt. Die gleiche Prozedur wird in dieser Weise erneut an der nun letzten Position vollzogen. Der Programmausschnitt für diese Bearbeitung sieht dann so aus, wenn wir den Index für das Zeichenarray "i" nennen:

```
do
  s[i++] = n % 10 + '0';
while((n /= 10) > 0);
```

Wie Sie sehen, wird solange die letzte Stelle umgewandelt und in "s" gespeichert, bis unsere Zahl, die in "n" abgelegt wurde, durch das ständige Dividieren bei 0 angelangt ist. Ach ja, das Vorzeichen dürfen wir nicht vergessen, da es sonst in unserer Schleifenbedingung (Zahl größer 0) Schwierigkeiten bereiten würde. Am einfachsten ist es, wenn wir vor der Umwandlung die Zahl positiv machen und gegebenenfalls einen Merker für einen

negativen Wert setzen. Nach der kompletten Umwandlung erhält die Zeichenkette dann auch ihr Minuszeichen zurück.

So, wie die Sache aber bisher läuft, erhalten wir nach der kompletten Bearbeitung aus der Zahl 123 den String "321", da ja immer nur die letzte Stelle bearbeitet und im String abgelegt wird. Die Lösung dieses Problems ist recht einfach: Wir schreiben eine weitere Funktion, die einen String in sich umdreht. Gehen wir also nun davon aus, daß wir bereits eine solche Funktion haben, so müßte unsere Routine so aussehen:

```
/*********************************************/
/* Name:       itoa                          */
/* Parameter: n (int), s (String)            */
/* Funktion:  Integer in String umwandeln    */
/* Sonstiges: benötigt reverse()             */
/*********************************************/

#define EOS '\0'
#define FALSE 0
#define TRUE 1

itoa(n, s)
register int n;
register char *s;
{
  register int i = 0;
  register int vorzeichen = FALSE;

  if(n < 0)
    {
      vorzeichen = TRUE;
      n = -n;
    }
  do
    s[i++] = n % 10 + '0';
  while(n /= 10);
  if(vorzeichen)
    s[i++] = '-';
  s[i] = EOS;
  reverse(s);
}
```

In den recht umfangreichen Vorspann habe ich ein paar wichtige Informationen hineingeschrieben. Die von uns entwickelten Funktionen sollen ja, wenn sie fertiggestellt sind, auch stets gebrauchsfähig bereit stehen. Nach einiger Zeit wird man wohl den einen oder anderen Funktionsnamen und deren Übergabeparameter vergessen haben. Dann braucht man nur in den Vorspann mit den Bemerkungen zu sehen, und schon ist man wieder im Bilde. Diese Funktionen können unabhängig von anderen Funktionen compiliert werden und, falls Ihr Compiler dies erlaubt, in eine Bibliothek aufgenommen werden. Selbstverständlich können die Sourcedateien auch in Ihre aktuelle Datei mit

```
#include "itoa.c"
```

eingebunden werden, was allerdings die Compilierzeit erhöht. So viel zum Thema der Dokumentierung.

Weil diese Funktion in eine Library aufgenommen werden soll, soll sie auch wirklich auf dem aktuellsten Stand des Wissens sein. Bevor sie also dort verschwindet, sollte aus der Funktion in bezug auf Geschwindigkeit auch das letzte herausgeholt werden. Bei unserer itoa-Routine läßt sich das durch die Definition aller Variablen als register-Variablen durchführen. Die für diese Funktion nötigen Defines lassen wir auch nicht aus, auch wenn es vielleicht als umständlich erscheint, ein Define für eine einzige Anwendung festzulegen. Es verbessert allerdings den Lesefluß, da Sie in eigenen, meist erheblich längeren Programmen stets zu diesen Makros greifen.

17.3 reverse

Nun noch zu der übergangenen Funktion "reverse", die einen übergebenen String umdrehen soll. Der Aufbau dieser Routine stellt kein Problem für uns dar. Wir benötigen zwei Pointer oder Indices, die auf den Anfang und das Ende des Strings gesetzt sind. Diese tauschen jeweils ihr Element untereinander aus und werden danach aufeinander zubewegt (Der Zeiger am Anfang wird erhöht, der am Ende erniedrigt). Es wird solange getauscht,

bis sich beide Zeiger erreicht haben, d.h. sie haben sich bereits gekreuzt oder zeigen auf das gleiche Element. Mit einem schmucken Funktionskopf haben wir dann unsere Routine komplettiert.

```
/*********************************************/
/* Name:       reverse                       */
/* Parameter:  s (String)                    */
/* Funktion:   String umdrehen               */
/* Sonstiges:  benötigt strlen()             */
/*********************************************/


reverse(s)
register char *s;
{
  register int c, i, j;

  for(i = 0, j = strlen(s) -1; i < j; i++, j--)
    {
      c   = s[i];
      s[i] = s[j];
      s[j] = c;
    }
}
```

Um den Index "j", der auf das letzte Element von "s" zeigen soll, zu initialisieren, wird die strlen-Funktion benutzt. Diese muß in jedem C-Compiler-Paket mitgeliefert werden, sonst können Sie unsere, bereits im vorherigen Kapitel definierte strlen-Funktion verwenden. Beide Routinen (itoa und reverse) sollten Sie in einer Datei mit dem Namen "itoa.c" speichern, da wir später noch einmal darauf zugreifen wollen. Nebenbei sei angemerkt, daß auch die atoi-Routine zur Standardausrüstung eines jeden C-Compilers gehört. Um allerdings etwas wirklich Neues zu schreiben, müßte man entweder eine sehr komplizierte Aufgabenstellung benutzen oder aber eine sehr spezielle Routine programmieren. Standardfunktionen, die man fast täglich einsetzt, haben bereits andere vor uns geschrieben. Da wir die Sprache C aber anhand praktischer Beispiele erlernen möchten, hoffe ich, daß Sie sich nicht unterfordert fühlen.

18. C-Features

In diesem Kapitel werden einige Komponenten von C betrachtet, die Ihnen in anderen Sprachen wohl kaum unterkommen werden. Teilweise erlauben gerade sie, die Vorteile von C bezüglich Geschwindigkeit und Flexibilität erst voll zu nutzen.

18.1 ?:-Operator

Ein spezieller C-Operator, den Sie in anderen Sprachen wohl vergeblich suchen, lautet "?:". Der "?:"-Operator wertet das erste Statement aus und liefert dann, falls der Ausdruck wahr ist, den darauf folgenden Ausdruck. Sollte der ausgewertete Ausdruck falsch (=0) sein, so wird der zweite, dem Doppelpunkt folgende Ausdruck übergeben. Der Operator wird in der Form:

ergebnis = (ausdruck1) ? (ausdruck2) : (ausdruck3);

verwendet. Sollte "ausdruck1" ungleich Null sein, so wird "ausdruck2" an "ergebnis" übergeben, sonst erhält es "ausdruck3" zugewiesen. Ein konkretes Beispiel:

```
c = (a>b) ? a : b;
```

Das wäre vergleichbar mit der if-Konstruktion:

```
if(a > b)
  c = a;
else
  c = b;
```

Dieser Term liefert das Maximum von a und b. Da dies sehr einfach zu formulieren ist, werden Minimum- und Maximum-Funktion meist mit dieser Operation bestimmt. Als Defines:

```
#define MIN(a,b) (((a)<(b))?(a):(b))
#define MAX(a,b) (((a)>(b))?(a):(b))
```

Werfen Sie doch noch einmal einen Blick in "stdio.h" (So ziemlich am Ende). Dort finden wir auch unsere Definitionen wieder.

18.2 sizeof

Ein weiterer Operator ist sizeof (size of, engl. Größe von), dessen Funktion durch die bloße Übersetzung schon ziemlich klar ist. Er liefert die Größe eines Objektes, also einer Variable, egal welchen Types. Die Einheit, die dieser Operator zurückgibt, ist auf der Basis von char-Elementen definiert, d.h. aus

```
char zeichen = 'a';
```

folgt sizeof(zeichen) ist 1.

In der Praxis ist das Ergebnis aber stets die Anzahl der belegten Bytes für das untersuchte Objekt. Mit dem folgenden kurzen Programm können Sie feststellen, wieviel Platz die verschiedenen Datentypen bei Ihrem Compiler verbrauchen, also auch die Frage beantworten, ob eine int-Variable 2 oder (wie beim Lattice) 4 Byte beansprucht.

```
main() /* Anzeige des fuer die Datentypen notwendigen Speicherplatzes */
{
  printf("\nDatentyp\tSpeicherplatz in Bytes\n");
  printf("char\t\t%d\n",   sizeof(char));
  printf("short\t\t%d\n",  sizeof(short));
  printf("int\t\t%d\n",    sizeof(int));
  printf("long\t\t%d\n",   sizeof(long));
  printf("float\t\t%d\n",  sizeof(float));
  printf("double\t\t%d\n", sizeof(double));
  printf("Zeiger\t\t%d\n", sizeof(char *));
}
```

18.3 Bitmanipulationen

Wir haben bereits alle Operatoren abgehandelt, bis auf die, welche einzelne Bits beeinflussen. Das wollen wir nun nachholen.

Außer den logischen Verknüpfungen gibt es noch Operatoren für die Bitverknüpfung. Ein Bit (Binary Digit) ist eine Stelle einer Binärzahl und kann deshalb nur die beiden Werte 0 und 1 annehmen. Die Umwandlung in das Binärsystem erfolgt analog zu Transformationen ins Oktal- oder Hexadezimalsystem (ein Umwandlungsprogramm steht uns ja bereits zur Verfügung). Das Bit ist auch gleichzeitig die kleinste Einheit, mit der der Rechner etwas anfangen kann. Es ist die Basis für alle anderen Größen, die im Computer verwendet werden können. Ein Byte z.B. besteht aus 8 Bit, ein Wort aus 16 Bit und ein Langwort aus 32 Bit. In char-Variablen kann, wie Sie schon wissen, ein Zeichen gespeichert werden. Hierfür wird ein Byte, also 8 Bit verwendet, so daß sich darin $2^8 = 256$ verschiedene Zahlen speichern lassen. Ein Integer-Wert enthält je nach Compiler 16 oder 32 Bit und ein long-Wert 32 Bit. Auf diese Datentypen kann nun auch auf jedes einzelne Bit zugegriffen werden. Es ist allerdings nicht möglich diese Operatoren bei float- oder double-Variablen zu benutzen.

18.3.1 UND

Der UND-Operator besteht aus dem "&"-Zeichen. Aber das ist doch der Adreß-Operator, müßte es Ihnen sofort durch den Kopf schießen. Nun, das ist gar nicht so einfach, da man dieses Zeichen für beide Zwecke einsetzen kann. Man muß also den Zusammenhang erkennen, in dem es verwendet wird. Steht es alleine vor einer Variablen, so stellt es den Adreß-Operator da. Wird es wie bei einer normalen arithmetischen Gleichung zwischen zwei Werte gesetzt, dann haben wir das binäre UND vor uns.

Passen Sie auf, die Begriffe "logisch" und "binär" sind nicht irgendein Füllsel, sondern dienen zur eindeutigen Unterscheidung zwei ganz verschiedener Operatoren! Das logische UND sieht etwas anders aus: "&&".

Durch UND (&) können einzelne Bits gelöscht werden. Ein gesetztes Bit hat den Wert 1, ein gelöschtes Bit den Wert 0. Das folgende Schema zeigt den Zusammenhang zwischen verschiedenen Bitkombinationen:

Verknüpfungstabellen

UND

&	0	1
0	0	0
1	0	1

ODER

|	0	1
0	0	1
1	1	1

EXOR

^	0	1
0	0	1
1	1	0

Ein Bit ist nach der UND-Verknüpfung nur dann gesetzt (1), wenn beide Bits gesetzt waren, sonst ist das Ergebnisbit 0. Vergleichbar wäre dies mit

```
if(bit1 == 1 && bit2 ==1) /* Hier steht das logische UND! */
  erg_bit = 1;
else
  erg_bit = 0;
```

18.3.2 ODER

Mit dem ODER-Operator (|) können gezielt Bits gesetzt werden. Auch hier hilft wieder ein kleiner Blick in obige Tabelle für die verschiedenen Bitkombinationen.

Bei der ODER-Verknüpfung ist das resultierende Bit gesetzt, wenn ein oder alle beide Bits gesetzt waren. Nur falls beide Bits 0 enthalten, lautet das Resultat der ODER-Verknüpfung auch 0. Zum Setzen einzelner Bits wird deshalb "|" verwendet, während zum Löschen "&" verwendet wird.

Die zu setzenden Bits werden in einer sogenannten Maske gespeichert. Eine Maske stellt eine Zahl dar, die anschaulich betrachtet, über den zu bearbeitenden Wert gelegt wird. Wird nun eine Variable durch ODER mit dieser Maske verknüpft und das Ergebnis wieder in dieser Variablen abgelegt, so sind alle in der Maske gesetzten Bits nun auch in der Variablen gesetzt.

Beispiel: In der Variablen "flags" soll das Bit Nr. 2 (man beginnt beim Zählen auch hier mit 0) gesetzt werden:

```
#define MASKE 4
int flags = 73;

flags |= MASKE;
```

Durch das ODERn mit dem Wert "2^2 (Nummer des zu setzenden Bits)" = 4 besitzt die Variable "flags" nach dieser Operation auf jeden Fall ein gesetztes 2. Bit.

Zum gezielten Löschen bestimmter Bits werden die entsprechenden Bits der Maske auf 0 gesetzt. Durch die UND-Verknüpfung erhält man im Ergebnis die gewünschten Nullbits. Beispiel: Bit 1 und 4 sollen gelöscht werden.

```
int flags = 37;

flags &= 0355; /* Alle Bits außer 1 und 4 sind hier gesetzt (0-7) */
```

Jedes Bit hat seine eigene Wertigkeit, die sich nach der Priorität richtet. So hat beispielsweise das Bit mit der Nummer 3 die Wertigkeit 8 (2^3). Eine Tabelle mit den einzelnen Wertigkeiten der Bits:

Bitnummer	0	1	2	3	4	5	6	7
Wertigkeit	1	2	4	8	16	32	64	128

Einige Beispiele für Bitverknüpfungen:

```
1 & 2 = 0
2 & 6 = 2
7 & 8 = 0
9 & 12 = 8
```

Das letzte Beispiel sehen wir uns noch einmal genauer im Dualsystem an, wo es wesentlich leichter zu erfassen ist:

9 (dez) = 1001 (dual), 12 (dez) = 1100 (dual)

```
    1001
&   1100
    1000
```

1000 (dual) = 8 (dez)

Dasselbe Spielchen für die ODER-Verknüpfung:

```
1 | 2 = 3
2 | 6 = 6
7 | 8 = 15
9 | 12 = 13
```

Und konkret noch einmal am letzten Beispiel:

```
    1001
|   1100
    1101
```

1101 (dual) = 13 (dez)

Es ist sehr wichtig, die Zeichen "&" und "&&" fein säuberlich auseinanderzuhalten. "&" verknüpft die Ausdrücke bitweise, "&&" stellt nur einen logischen Vergleich an, aus dem entweder 1 (wahr) oder 0 (falsch) resultiert. So ist 2 & 1 = 0, aber 2 && 1 = 1.

Auch für die Operatoren "|" und "||" gilt diese Unterscheidung, da sonst besonders bei Abfragen (Schleifen und Bedingungen) ein sehr eigenartiges Verhalten festgestellt wird.

18.3.3 Bitgeschiebe

Weitere Operatoren zur Bitmanipulation sind >> und <<. Mit ihnen können die Bits innerhalb eines Feldes nach links oder rechts weitergeschoben werden. Ein Verschieben um eine Stelle nach links (<<) entspricht der Multiplikation mit 2, nur ist sie viel schneller zu berechnen. Das liegt an der Art und Weise, wie Zahlen und Daten im Rechner abgelegt sind und vom Prozessor bearbeitet werden. Ein Verschieben nach rechts gleicht einer Division durch 2. Je nach Datentyp werden entweder Nullbits oder gesetzte Bits an die freie Stelle nachgeschoben. Bei unsigned-Werten werden in jedem Falle Nullbytes nachgeschoben, während bei "normalen" int-Werten (mit Vorzeichen!) es nicht eindeutig festgelegt ist. Bei positiven Zahlen sollten Nullbits links eingefügt werden, bei negativen Zahlen gesetzte Bits. Darauf ist aber kein Verlaß.

Die Anzahl, um wieviel Stellen die Bits verschoben werden sollen, wird hinter dem Operator angegeben.

5 << 3 = 40

101 (dual) um 3 Stellen nach links verschieben (es werden stets ungesetzte Bits nachgeschoben): 101000 (dual) = 40 (dez).

Zum Umrechnen von Dezimalzahlen in das Dualsystem können Sie unser Programm aus dem vorherigen Kapitel verwenden.

Zur Bitmanipulation können noch weitere Operatoren eingesetzt werden, die kaum noch Wünsche offen lassen. Außer den bereits demonstrierten UND und ODER gibt es noch die EXKLUSIVE-ODER-Verknüpfung und den Komplement-Operator.

18.3.4 EXKLUSIVE-ODER

Der EXKLUSIVE-ODER-Operator "^" ist, wie der Name andeutet mit dem ODER-Operator verwandt. Der einzige Unterschied besteht in dem Fall, daß beide Bits gesetzt sind. Bei der ODER-Verknüpfung resultiert daraus ein ebenfalls gesetzes Bit, bei der EXKLUSIVE-ODER-Verknüpfung ist das Ergebnis aber ein ungesetztes Bit. Die Tabelle zur EXKLUSIVE-ODER-Verknüpfung lautet demnach:

EXOR	0	1
0	0	1
1	1	0

z.B. 2 ^ 1 = 3

"^" bitte nicht mit dem Hochpfeil für die Potenzierung verwechseln, so etwas gibt es in C nicht!

Zum Merken: Das Bit wird nur gesetzt, wenn beide Bits verschieden waren.

18.3.5 Komplement-Operator

Der Komplement-Operator "~" benötigt lediglich einen Parameter. Bei diesem werden dann alle Bits umgedreht. Aus gesetzten werden ungesetzte Bits und umgekehrt. Es ist empfehlenswert, diesen Operator nur bei Variablen anzuwenden, die als "unsigned" definiert wurden, da sonst das Vorzeichen ebenfalls beeinflußt wird.

```
unsigned zahl = ~3;
```

In der Variablen sind nun alle Bits außer den ersten beiden (Priorität 0 und 1 = 1 + 2 = 3) gesetzt, so daß die Variable nun die folgende Bitfolge enthält (ausgehend von 16-Bit-Integer)

1111 1111 1111 1100

= 65532

18.4 goto

Als wirklich letzter C-Befehl bleibt noch der goto-Sprung übrig. Vielleicht kennen Sie diesen Befehl z.B. von BASIC, aber in C ist er wirklich eine Besonderheit. Die Aufführung dieses Statements am Ende der Liste kann durchaus als Wertung der Priorität verstanden werden, die diesem Befehl beigemessen wird. Dieser Befehl ist in C ziemlich verpönt, da er die sonst so vorbildliche strukturierte Programmierung zunichte machen kann. Durch wildes Hin- und Herspringen in einer Funktion geht der Überblick völlig verloren, BASIC-Programmierer können ein Lied davon singen. Trotzdem ist der goto-Befehl in C nicht völlig sinnlos. Besonders bei der Behandlung von Fehlern kann er gewinnbringend eingesetzt werden. Tritt innerhalb mehrerer Schleifen ein Fehler auf, der ein Weiterarbeiten unmöglich macht, so kann nur mit der goto-Anweisung entkommen werden. Die sonst hierfür benutzte break-Anweisung kann immer nur eine Schleife abbrechen, nicht aber mehrere gleichzeitig. (Man könnte natürlich auch mit einigen Abfragen und diversen break-Anweisungen alle Schleifen abbrechen. Das wird allerdings so unübersichtlich und aufwendig, daß man doch eher zum "goto" greift).

Die Verwendung des goto-Statements setzt natürlich ein Label, eine markierte Zeile voraus, die durch goto angesprungen werden soll.

```
label: printf("Hier springt goto hin!\n");
  ...
  if(fehler)
    goto label;
```

Solche Labels können in beliebiger Zahl im Programmtext definiert werden und erhalten einen Doppelpunkt nachgestellt. Man benötigt sie aber nur für den goto-Befehl, und sie werden genau wie Variablennamen gebildet.

Bemerkung: Das Label und der Sprungbefehl müssen in der gleichen Funktion verwendet werden. Es ist also nicht möglich, willkürlich quer durch alle Funktionen zu springen.

19. Zusammengesetzte Datentypen

Nachdem wir alle wichtigen Befehle behandelt haben, kommen wir zu den Bonbons von C. Dazu gehören Datentypen, die Sie selbst nach Ihren Anforderungen zusammenbasteln können. Am besten, Sie sehen sich so eine Definition einmal an:

19.1 Eine Struktur

```
struct {
        char vorname[20];
        char nachname[30];
        int alter;
        double einkommen;
        int geschlecht;
      } person;
```

Hierdurch wird eine Variable namens "person" definiert. Diese besteht aus mehreren Teilvariablen, die innerhalb der geschweiften Klammern näher bestimmt werden. Für den Vornamen sind 20, für den Nachnamen 30 Zeichen reserviert. Dazu kommen noch Felder für Alter, Einkommen und Geschlecht. Ähnlich wie bei Array haben wir viele Einträge unter einem Namen zusammengefaßt. Der Unterschied liegt nun darin, daß nicht mehr gleichartige, sondern verschiedenartige Variablentypen innerhalb der Struktur auftauchen. Um auf die einzelnen Teile dieser Variablen zugreifen zu können, muß sie noch weiter spezifiziert werden. Bei Arrays genügte der Index, mit dem wir hier überhaupt nichts anfangen können. Bei der Struktur geschieht das durch den "."-Operator (Punkt) oder mittels "->". Eine Zuweisung von 30 an das Element "alter" sieht so aus:

```
person.alter = 30;
```

Einfach, nicht? Genauso können alle anderen Felder angesprochen werden:

```
person.geschlecht = 0;
person.einkommen = 3000.0;
strcpy(person.vorname, "Peter");
strcpy(person.nachname, "Silie");
```

Möchte man lieber mit Zeigern auf eine solche Konstruktion hantieren, muß man den Datentyp angeben. Diesen haben wir noch gar nicht verallgemeinert, sondern einfach eine komplette Variable zusammengebastelt. Es fehlt also ein Name wie z.B. "int" oder "float", über den wir dann auch weitere Variablen wie einen Zeiger darauf definieren können. Sollen mehrere solcher Variablen oder eben Zeiger darauf verwendet werden, so empfiehlt es sich, einen neuen Datentyp zu kreieren, der dann auch einen eigenen Namen erhält. Dies wird einfach dadurch erledigt, daß nach dem struct-Befehl der Typname angegeben wird. Nennen wir ihn "Person", mit einem Großbuchstaben zu Beginn, um anzuzeigen, daß es sich nicht um eine Variable handelt. Das ist nur ein Tip, keine Vorschrift. Da wir uns geeinigt haben, Variablen und Funktionen in Klein- und Defines in Großbuchstaben darzustellen, verwendet man bei Strukturen gerne die gemischte Schreibweise.

```
struct Person {
                char vorname[20];
                char nachname[30];
                int alter;
                double einkommen;
                int geschlecht;
              } person;
```

Nun kann durch

```
struct Person *zeiger;
```

ein Zeiger auf eine solche Datenstruktur initialisiert werden. Um damit auf ein Element der Struktur zugreifen zu können, wäre der Ausdruck

```
(*zeiger).alter = 30;
```

nötig (Die Klammern müssen wegen der höheren Priorität des "."-Operators gesetzt werden). In der Praxis wird aber ein spezieller Verweisoperator verwendet, der aus einem Minus- und einem Größerzeichen besteht.

```
zeiger->alter = 30;
```

ist das Pendant zur obigen Anweisung mit dem "."-Operator, wird aber (unter anderem) wegen der ersparten Tipparbeit und der Übersichtlichkeit vorgezogen. Außerdem kann man sich doch durch den stilisierten Pfeil den Satz "...zeigt auf..." viel besser vorstellen. Oder?

Mit der neugeschaffenen Struktur können Sie alle Operationen ausführen, die auch mit Basisdatentypen möglich sind, z.B. Vektoren, also Arrays definieren:

```
struct Person bewohner[100], *bew_poi;
```

Dadurch hat man nun 100 solcher Strukturen, in denen jeweils die angegebenen Teilvariablen enthalten sind. Das Ansprechen der einzelnen Einträge kann entweder mit einem Index erfolgen (bewohner[3].gehalt = 2500.0), oder nach der Initialisierung des Zeigers mit

```
bew_poi = bewohner;
```

Bemerkung: Sie erinnern sich, daß der Name die Adresse des ersten Elementes darstellt. Alternativ könnte

```
bew_poi = &bewohner[0];
```

geschrieben werden. Mit dem Zeiger kann nun in der gleichen Weise mit den Einträgen umgesprungen werden, wie Sie es schon gewohnt sind (zeiger->gehalt = 2500.0).

Mit dem Zeiger können Sie auch das gesamte Array durchkämmen, beispielsweise durch

```
while(zeiger->alter)
  zeiger++; /* durchkämmt alle Einträge */
```

Damit erhält man, unter der Voraussetzung, daß in einem unbenutzten Eintrag der Wert Null im Alter untergebracht ist, den Zeiger auf das erste freie Element gesetzt. Zu weiteren Anwendungen der struct-Anweisung kommen wir später.

19.2 Bitfelder

Der letzte übriggebliebene Datentyp sind Bitfelder. Bitfelder sind eigentlich eine Art Strukturdefinition. Nur werden hier keine bestehenden Datentypen zusammengesetzt, im Gegenteil, sie werden quasi demontiert. Man definiert eine Variable, die aus einer bestimmten Anzahl von Bits besteht, also stets ganze Zahlen darstellt. Der Wertebereich ist von der verwendeten Bitanzahl abhängig und kann mit der Formel $2^{\text{Bitanzahl}}$ errechnet werden. Diese Felder werden in int-Objekten eingerichtet, so daß die maximale Feldbreite 16 Bit beträgt. Das gilt auch beim Lattice-C, der ja sonst beim Begriff "int" stets eigene Wege geht. Paßt ein Feld nicht mehr in einen teilweise belegten Integer-Wert, so wird er in den nächsten eingetragen. Durch geschickte Wahl der Feldbreite, läßt sich allerhand Speicherplatz sparen.

```
struct {
        unsigned geschlecht : 1;
        unsigned verheiratet : 1;
        unsigned anz_kinder : 4;
      } daten;
```

Genau wie bei anderen Strukturdefinitionen sind innerhalb der geschweiften Klammern die Datentypen untergebracht. Um auch sicherzugehen, daß es sich bei einem Bitfeld um vorzeichenlose ganze Zahlen handelt, verwendet man "unsigned", eine Abkürzung für "unsigned int". Nach dem Namen dieses Feldes wird die Feldbreite in Bits durch einen Doppelpunkt getrennt. Obige Definiton belegt deshalb 2 Byte (Größe eines 16-Bit-Integers), ist aber noch längst nicht ausgelastet. Da erst 6 Bits belegt sind (1 + 1 + 4), können noch 10 weitere Bits für andere Aufgaben vergeben werden, ohne daß hierfür zusätzlicher Speicherplatz beansprucht wird.

```
struct {
        unsigned geschlecht : 1;
        unsigned verheiratet : 1;
        unsigned anz_kinder : 4;
        unsigned alter : 7;
        unsigned anz_kfz : 3;
      } daten;
```

Hier sind zusätzliche Daten untergebracht worden, die komplett auch nur 2 Byte belegen. Der Zugriff auf jedes Bitfeld erfolgt wieder mit dem "."-Operator:

```
daten.anz_kinder = 2;
```

Die begrenzten Wertebereiche sind aber unbedingt zu beachten, so daß bei einer solchen Definition keine Familie auftauchen darf, die mehr als 15 (2^4-1) Kinder hat oder mehr als 7 (2^3-1) Autos fährt.

19.3 union

In C existiert noch eine Spezialvariable, die alle erdenklichen Datentypen aufnehmen kann. Eine Variante (engl. union), so heißt diese Konstruktion, wird vom Compiler so dimensioniert, daß sie alle bei der Definition angegebenen Datentypen in sich speichern kann.

```
union Universa {
                int i;
                double d;
                struct Person;
                char c[100];
              } ergebnis;
```

Alle angegebenen Datentypen können in "ergebnis" gespeichert werden. Dabei ist es natürlich von Nutzen, wenn man sich merkt, welcher Typ gerade darin enthalten ist, z.B.

```
ergebnis = 2.8;
```

oder

```
strcpy(&ergebnis, c);
```

Der Speicherbedarf einer solchen Variablen richtet sich natürlich nach der Länge des größten Eintrages. Im obigen Beispiel wären dies 100 Bytes, die vom Array "c" belegt wird. Beachten Sie aber, daß immer nur ein Typ in dieser Variablen gespeichert werden kann. Für welche sinnvollen Aufgaben man union-Strukturen einsetzen kann, ohne das gleiche Problem auch mit den anderen C-Datentypen lösen zu können, ist mir bisher noch nicht eingefallen.

19.4 Aufzählung: enum

Durch das C-Wort enum kann man einen Datentyp definieren, der Variablen konstante Werte zuordnet. Dies sind stets Intergerzahlen, die etwa wie folgt Verwendung finden können:

```
/* Definition eines solchen Datentyps */
enum farbe (rot, gruen, blau, schwarz = 9, weiss)

/* Variablendefinition */

enum farbe var, *farb_ptr = &var;

var = blau;
if(*farb_ptr == gruen)
  *farb_ptr = schwarz;
```

Der Aufzählungstyp ordnet jedem Namen einen Integer-Wert zu, der bei Null beginnt und bei jedem folgenden Element um Eins erhöht wird. Durch direkte Zuweisung kann man auch Werte überspringen. Die Werte, die in "farbe" definiert wurden, sind:

rot	0
gruen	1
blau	2
schwarz	9
weiss	10

19.5 typedef

Für neue Datentypenamen kann die typedef-Anweisung herangezogen werden. Ein mit diesem Befehl festgelegter Name kann als weiterer Datentyp bei Definitionen verwendet werden.

```
typedef double FLOAT;
```

In dem folgenden Programmtext kann FLOAT anstatt des Datentyps "double" benutzt werden. Der Vorteil dieser Anweisung liegt darin, daß durch Änderung der typedef-Definition im gesamten Programm auch die entsprechenden Änderungen vollzogen sind, ähnlich wie bei Defines. Auch umfangreichere Datentypen können mit diesem Befehl abgekürzt werden:

```
typedef char * STRING;
```

Alle Zeiger auf char-Elemente können einfach durch das Symbol STRING definiert werden. Der Vorteil des typedef-Befehls im Gegensatz zur "#define"-Anweisung ist, daß die Definition des Ersatzes etwas anders vorgenommen wird. Wie Sie wissen, kann man einen Text durch einen anderen Text ersetzen. Ein Leerzeichen kennzeichnet dabei die Stelle, an der der Textersatz auftaucht. Die Definition des Types STRING wäre also nicht machbar, da alles nach dem Space hinter "char" bereits als Ersatztext gilt. Beim typedef-Behl ist dies genau umgekehrt, der letzte String "STRING" ist der Ersatz für den Datentyp "char *". Das oder die Leerzeichen, die mitten drin stehen, gehören immer zur Definition des darzustellenden Datentyps.

Damit wäre die Einführung der C-Schlüsselworte beendet, und wir können beginnen, die Befehle in kleineren Programmen zu nutzen. Dabei sollen die Kenntnisse gefestigt werden, denn nur Übung macht den Meister. Ich möchte Sie nochmals dazu anhalten, ruhig einmal auszuprobieren, was passiert, wenn Sie bei dem einen oder anderen Programm bestimmte Parameter ändern oder Programmteile hinzufügen.

20. Wichtige Grundbegriffe

20.1 Deklarationen

Was Deklarationen sind, wissen wir schon, nämlich die Bekanntmachung an das Programm, welcher Datentyp einer Variablen oder Funktion erwartet werden soll. Es wurden auch schon diverse Deklarationen verwendet, deshalb wollen wir etwas Systematik hineinbringen. Es können nämlich Ausdrücke auftauchen, bei denen man raten muß, welcher Datentyp deklariert wurde, wenn der Ausdruck nicht nach gewissen Regeln untersucht wird. Nehmen wir zuerst einige einfache Beispiele (die ersten drei sind auch Definitionen):

`int i;`	Integervariable
`float array[10];`	Floatvariable
`double *ptr;`	Zeiger auf double-Elemente
`long funk();`	Funktion, die einen long-Wert liefert

Jede Deklaration beinhaltet grundsätzlich einen elementaren Datentyp (char, int, float), der gegebenenfalls durch eine besondere Speicherklasse (auto, extern, register, static) oder durch Attribute wie long, short und unsigned ergänzt werden kann.

`extern double sin();`	Funktion aus einer anderen Datei liefert double-Wert
static short ziffer;	kleine Integervariable
`register long i;`	long-Variable soll in einem Register plaziert werden

Jeder Name kann zusätzlich mit diversen Kombinationen von *, [] oder auch () versehen werden. Dabei wird der Stern links vom Namen als Zeichen für einen Pointer, die Klammern rechts davon gesetzt. Die Klammern sollten bei einer Deklaration keine Werte enthalten, weil es sich sonst um eine Definition handelt ([] bei Arrays, () für Funktionen). Nimmt man sich nun alle Bausteine einer Deklaration zusammen, können komplizierte Konstruktionen erstellt werden.

`long *feld();`	Funktion, die Pointer auf long-Wert zurückgibt
`int *i_ptr[];`	Feld von Integer-Zeigern
`float (*berech)();`	Zeiger auf Funktion, die float-Werte zurückgibt
`char *(*text)();`	Zeiger auf Funktion, die Zeiger auf char liefert
`int *(*text_arr[])();`	Array von Pointern auf Funktionen, die Zeiger auf int liefern

Na, verstehen Sie davon noch eine Deklaration. Zumindest die ersten beiden könnten Ihnen noch einleuchten. Aber dann...? Keine Angst vor solchen komplizierten Ausdrücken, alle werden nach einheitlichen Regeln gebildet. Geht man sie Schritt für Schritt durch, so ist es später nur noch Routinearbeit solche Zusammenstellungen zu entschlüsseln. Dazu benötigt man wieder die Tabelle der Operatorenprioritäten, die sich im Anhang befindet. Daraus entnimmt man, daß die runden Klammern mehr als der Pointer binden. Deshalb handelt es sich bei der Variablen "feld" erst einmal um eine Funktion. Nun wenden wir uns wieder der anderen, linken Seite zu. Dort steht der Stern, der den Ausdruck nun als Funktion definiert, die einen Zeiger liefert. Nun wird wieder auf die andere Seite gewechselt, auf der keine zusätzlichen Informationen mehr zu finden sind (Die runden Klammern waren bereits die letzten bearbeiteten Zeichen auf dieser Seite). Daraufhin springen wir wieder auf die linke Seite, wo wir den Datentyp long finden. Das war die letzte Information. Zusammengefaßt haben wir dadurch eine Funktion, die einen Pointer auf den Typ long liefert.

Es ist stets nach der Bearbeitung der einen Seite die folgende Information auf der anderen Seite zu suchen, falls dies die Prioritäten zulassen. Dies noch einmal am letzten, dem wohl komplexesten Beispiel:

```
int *(*text_arr[])();
```

Sie brauchen hier nicht die Hände über dem Kopf zusammenzuschlagen! Wenn wir strikt unserer vorherigen Vorgehensweise

folgen ist auch dieser Ausdruck leicht zu entschlüsseln, nur wird halt die Beschreibung einiges länger.

Wir beginnen mit dem Namen "text_arr". Jetzt prüfen wir, welche Operatoren der Priorität nach zuerst ausgeführt werden müssen (nur rechts oder links des Namens). Dies sind die eckigen Klammern, die ein Array kennzeichnen. Somit haben wir den ersten Operatoren von der rechten Seite bearbeitet, dann gehen wir jetzt auf die linke. Dort steht der Pointer, so daß wir jetzt schon wissen, daß vor uns ein Array mit Pointern steht. Auf die andere Seite gewechselt, erfahren wir, daß die Pointer auf Funktionen zeigen sollen (Die Klammern sind wegen der höheren Priorität der Klammern gegenüber dem Stern nötig). Erneut gewechselt, sehen wir, daß diese Funktionen wiederum Zeiger zurückgeben. Da auf der rechten Seite keine Informationen mehr zur Verfügung stehen, machen wir auf der linken Seite mit dem Datentyp weiter. Dort wird noch festgelegt, daß die Pointer auf Integer zeigen. Zusammengefaßt haben wir es mit einem Array von Pointern auf Funktionen zu tun, die Zeiger auf int liefern. Komplizierter Ausdruck, komplizierter Satz, aber einfach zu ermitteln!

Wie Sie gesehen haben, sind Ihrer Phantasie kaum Grenzen gesetzt. Lediglich die Datentypen, die auch nicht bei Definitionen als Übergabewerte verwendet werden dürfen, sind unzulässig. So können z.B. keine Funktionen deklariert werden, die Strukturen, Arrays oder Funktionen selbst übergeben sollen. Zeiger auf solche Objekte sind aber zugelassen und auch die einzige Möglichkeit, wie Sie aus der Praxis mit Strings wissen, an diese Informationen heranzukommen.

Anmerkung: Neuere Compiler erlauben auch die Übergabe von Strukturen. Das ist aber von Compiler zu Compiler sehr verschieden. Der Ausdruck

```
&struktur
```

ist stets die Adresse der Struktur, aber

```
struktur
```

kann verschiedene Dinge darstellen. Bei den älteren Compilern ist dies genau wie der Ausdruck mit dem Adreß-Operator die Startadresse. Übergibt man aber "struktur" einem Compiler, der bereits Datenstrukturen übertragen kann, so werden nicht nur 4 Byte, die den Zeiger darauf darstellen, sondern das gesamte Datenfeld wird der aufgerufenen Funktion zur Verfügung gestellt.

20.2 Initialisierungen

Auch dieser Audruck ist uns bekannt. Aber genau wie bei der Deklaration ist hier noch einiges mehr zu sagen, als wir bisher benötigten. Deshalb noch eine kurze Wiederholung, bevor wir uns den Initialisierungen von Strukturen zuwenden.

Mit diesem Ausdruck bezeichnet man üblicherweise die erste Wertzuweisung einer Variablen. Vor Benutzung einer Variablen muß stets ein definierter Wert enthalten sein, da Sie sonst bei Berechnungen nicht nur unsinnige Ergebnisse erhalten, sondern auch Systemabstürze riskieren (nicht initialisierte Pointer). Die Initialisierung kann zum einen durch direkte Zuweisung in der Form

```
int i;
i = 0;
```

oder bereits in der Definition erfolgen:

```
int i = 0;
```

Die Initialisierung in der Definition hat den Vorteil, daß keine zusätzlichen Anweisungen mehr nötig sind, um die Variablen zu belegen. Das spart Zeit und Speicherplatz. Deklarationen, Definitionen und Initialisierungen können bei einem Datentyp auch beliebig gemischt werden.

```
double zahl, pi = 3.1415926, sin();
```

C erlaubt bei Initialisierungen beliebige Konstanten und Ausdrücke. So können in einer Zeile durchaus folgende Zuweisungen zu finden sein:

```
long zahl = x * pi - abs(y);
char *cp = string + strlen(string);
```

Dabei ist aber zu beachten, daß die verwendeten Variablen bereits selbst initialisiert wurden, da ansonsten auch "zahl" einen nicht definierten Wert enthält.

Bei Arrays oder Strukturen können geschweifte Klammern die Übersicht verbessern, um die einzelnen Einträge voneinander abzusetzen. Vor und hinter die gesamten Daten, die in die Variablen übertragen werden sollen, muß allerdings ein Paar geschweifte Klammern plaziert werden. Nach den jeweiligen Feldern folgt ein Komma, auch dann, wenn Sie die geschweiften Klammern verwendet haben. Nach der Initialisierung kommt stets ein Semikolon, das oft vergessen wird. Spätestens, wenn der Compiler anfängt, an dieser Stelle herumzumäkeln, werden Sie diesen Fehler entdecken. Einige Beispiele für korrekte Strukturdefinitionen:

```
struct KFZ {
             char marke[16];
             int kw;
             int tueren;
             double preis;
           };

struct KFZ will_haben =
{
  "BMW",
  120,
  4,
  40000.0
};
```

Oder auch zusammengefaßt bei der Strukturdefinition:

```
struct KFZ {
            char marke[16];
            int kw;
            int tueren;
            double preis;
          } will_haben =
{
  "BMW",
  120,
  4,
  40000.0
};
```

Die Beispiele für mehrdimensionale Array-Initialisierungen wurden bereits im Kapitel über Arrays und Zeiger besprochen. Dort wurde aber auch auf die Einschränkungen bezüglich der Speicherklassen hingewiesen. Initialisierungen sind nur bei globalen, externen oder statischen Variablen zugelassen. Soll trotzdem innerhalb einer Funktion eine auto-Variable auftauchen, die dieser Initialisierung entspricht:

```
char meldung[] = "Denken Sie an die Initialisierung!";
```

so können Sie mit einer Pointer-Definition diese Klippe umschiffen.

```
char *meldung = "Denken Sie an die Initialisierung!";
```

Sie unterliegen keinerlei Einschränkung, wenn Sie die untere Definition als Zeigervariable verwenden. Eine andere Möglichkeit ist eben die Verwendung von static-Variablen. Ob beispielsweise der String, den Sie darin abspeichern, in einer auto- oder static-Variablen abgelegt wird, kann Ihnen doch ziemlich gleichgültig sein. Nur ein kleines Wort, aber eine große Wirkung:

```
static char meldung[] = "Denken Sie an die Initialisierung!";
```

21. Zeiger-Arrays

Wir haben zuvor schon mit Zeigern und auch mit Arrays gearbeitet. Wie die Überschrift aber schon andeutet, kann man auch beide kreuzen, also ein Array mit Zeigern aufbauen. Stellen wir uns nun die Frage, wofür man so etwas braucht. Soll ein Programm nach seiner Fertigstellung auch ins Ausland verkauft werden, so müßte jeder einzelne Text im ganzen Programm gesucht und an dieser Stelle übersetzt werden. Einfacher und sicherer ist es, wenn an einer bestimmten Stelle im Programm oder vielleicht sogar in einer anderen Datei alle Texte aufbewahrt werden, die sich der Übersetzer dann zu Gemüte führen kann.

Versuchen wir uns doch einmal an einer Liste von Fehlermeldungen, die dem Benutzer bei Fehleingaben gezeigt werden sollen. Dazu ist die Verwendung von bestimmten Fehlernummern sinnvoll, da einige Fehler bestimmt an mehreren verschiedenen Stellen gemacht werden können. Bei dem jeweiligen Programmteil genügt dann ein Aufruf der Fehlerroutine, der die Fehlernummer übergeben wird. Den Rest sollte diese Funktion ausführen.

Eine mögliche Lösung dieses Problems wäre eine Funktion, die durch if-oder switch abfragt, ob dieser oder jener Fehler aufgetreten ist, und gegebenenfalls den Text ausgibt:

```
fehler(f_nummer)
int f_nummer;
{
  switch(f_nummer)
    {
      case 0:
        puts("Alles Ok, kein Fehler!");
        break;
      case 1:
        puts("Falsche Taste gedrückt!");
        break;
      case 2:
        puts("Bitte Diskette einlegen!");
        break;
```

```
        default:
          puts("Unbekannter Fehler ist aufgetreten!");
      }
  }
```

Nicht nur, daß diese Funktion recht umständlich formuliert ist, bei weiteren Fehlermeldungen ist jedesmal ein weiterer Funktionsaufruf von puts nötig. Die Funktion PutString ist nur dazu ausgelegt, einen String auf dem Bildschirm auszugeben, hat also nicht die Fähigkeiten wie printf. Dadurch ist sie aber einiges schneller als die universellen Ausgabefunktionen. Außerdem müssen diverse case-Anweisungen eingefügt werden, die das Programm nicht gerade schneller werden lassen. Da jeder Nummer eine bestimmte Fehlermeldung (String) zugeordnet werden kann, wäre es doch möglich, die Fehlernummer als Index auf ein Feld mit Strings zu benutzen. Da ein String in der Regel in "char fehler[81]" Platz findet, definieren wir den Platz für den Text als zweidimensionales Array:

```
char fehler[32][81];
```

Dadurch haben wir nun Platz für 32 Strings mit einer maximalen Länge von 81 Zeichen. Diese Formulierung ließe zwar die folgende kurze Routine zur Ausgabe von Fehlermeldungen zu,

```
fehler(f_nummer)
int f_nummer;
{
  puts(fehler[f_nummer]);
}
```

aber (das mußte ja kommen!) die Arbeit, die wir uns hier gespart haben, muß woanders erledigt werden. Die Initialisierung der einzelnen Strings muß ja mit der strcpy-Funktion erfolgen, so daß dann irgendwo dieser Ausschnitt zu finden ist

```
strcpy(fehler[0],"Alles Ok, kein Fehler!");
strcpy(fehler[1],"Falsche Taste gedrückt!");
strcpy(fehler[2],"Bitte Diskette einlegen!");
...
```

Ob wir also strcpy oder puts mit allen Texten aufrufen, ist also gehüpft wie gesprungen. Noch ein weiterer Nachteil soll hier genannt werden: der verschenkte Speicherplatz. Durch die Definition erhält jede Fehlermeldung 81 Zeichen (ein Nullbyte darunter) zur Verfügung gestellt, auch wenn diese nur 20 Zeichen wirklich benötigen. Beim AMIGA fällt dies kaum ins Gewicht, man sollte sich aber frühzeitig diesen Stil abgewöhnen. Denn wenn Sie das gleiche Spielchen für eine Textverarbeitung durchführen wollen, bei der jedes Wort einen Eintrag (vielleicht auf 60 Zeichen pro Wort beschränkt) bekommt, so können Sie aufgeben, bevor Sie angefangen haben. Der Speicher ist im Nu voll, ohne aber sinnvoll genutzt zu werden.

Hier treten nun in letzter Rettung die String-Arrays hervor, die (fast) alle diese Probleme beseitigen. Die Definition eines solchen char-Arrays würde lauten:

```
char *fehler[32];
```

Dieser Zeiger kann nun auf den Beginn einer Fehlermeldung gesetzt werden und macht diese dadurch unabhängig von festen Array-Längen. Der folgende String kann direkt hinter dem letzten Zeichen des Vorgängers beginnen, ohne die Angelegenheit zu komplizieren. Aber auch die Sache mit der Initialisierung ist umgangen. Wenn Sie sich noch erinnern können, wurde im Kapitel über Pointer ein Programm vorgestellt, das einen Zeiger auf einen String setzt, der sich im Programmtext befindet. Dies kann bereits bei der Definition des Zeigers geschehen und deshalb wird auch gleich das gesamte Pointerarray mit den Anfangsadressen der Strings initialisiert. Die Fehlermeldungen müssen in diesem Falle natürlich global definiert werden:

```
char *fehler[] =
{
  "Alles OK, kein Fehler!",
  "Falsche Taste gedrückt!",
  "Bitte Diskette einlegen!"
};
```

```
main()
{
  /* Alle Fehlermeldungen anzeigen */
  int i, fehler_anz = sizeof(fehler) / sizeof(char *);

  for(i = 0; i < fehler_anz; i++)
    printf("Fehlernr %d: \"%s\"\n", i, fehler[i]);
}
```

Da wir die Anzahl der Fehlermeldungen nicht zählen wollen, wird sie auch nicht bei der Definition mit angegeben. Dadurch können weitere Texte ohne weitere Änderungen zwischen die geschweiften Klammern eingetragen werden. Im eigentlichen Programm muß diese Anzahl erst noch berechnet werden. Dazu läßt man sich durch sizeof den belegten Speicherplatz von "fehler" geben. Unter dem Begriff "fehler" haben wir nun ein Array von Zeigern. Nun meldet uns sizeof (wie im obigen Fall), daß diese Variable 12 Bytes verbraucht. Das ist der Platzbedarf für alle Zeiger, der nichts mit dem Speicher für die Texte zu tun hat. Die 12 Bytes dividiert man durch den Platzbedarf eines char-Zeigers (und der ist 4 Bytes). Das Resultat ist die Anzahl der Zeiger, also der maximale Index minus Eins (mit 0 beginnen ja die Indices).

22. Nützliche Makros

Mit "#define" wurde bei uns schon reichlich gearbeitet. Der Ersatztext, auch Makro genannt, war aber immer sehr einfach gehalten. Es wurde ein Wort gegen ein anderes ausgetauscht. Das ist aber nur "halber Kram", denn man kann Makros sogar Parameter übergeben. Um Ihnen bei eigenen Gehversuchen etwas Hilfestellung zu geben, entwickeln wir mal einige nützliche Makros mit Parameterübergabe.

Funktionen, die recht wenig zu tun haben und ziemlich knapp formuliert werden können, bieten sich dazu an, als Makro formuliert zu werden. Warum? Nun, da Makros nur durch den Originaltext ersetzt werden, übersetzt der Compiler den C-Code direkt an dieser Stelle in Maschinensprache. Es sind keine Funktionsaufrufe nötig, denen eventuell noch Parameter übergeben müssen. Die nötigen Befehle stehen genau an dieser Stelle und nicht in irgendeinem Unterprogramm. Daher sind Makros in der Ausführung schneller und effizienter als Funktionsaufrufe, vergrößern bei häufiger Benutzung allerdings auch die Programmlänge. Denn die gleichen Operationen werden mehrfach identisch im Programmcode abgelegt, eben genau an der Stelle, an der sie benötigt werden. Ein ganz entscheidender Vorteil ist noch hervorzuheben: Makros sind in der Regel vom Datentyp unabhängig. Diesen Umstand wollen wir an dem Beispiel des Maximum-Makros betrachten:

```
#define MAX(a,b) ((a>b)? a : b)
```

Sind die Variablen i1 und i2 als Integer formuliert, so erhält man aus dem Makro-Ausdruck

```
erg = MAX(i1,i2);
```

den vom Präprozessor gelieferten Term:

```
erg = ((i1>i2)? i1 : i2);
```

Das Ergebnis ist also ebenfalls wieder ein Integerwert. Sind i1 und i2 aber als float-Variablen definiert, so entsteht zwar der

gleiche Ausdruck, als Ergebnis erhält man aber ebenfalls eine float-Zahl. Durch den Textersatz ist völlig egal, welcher Datentyp eingesetzt wird. Mit Funktionen ist das nicht möglich, da die Datentypen der Übergabeparameter explizit angegeben werden müssen. Dies stellt eine ernorme Vereinfachung bei der Benutzung dar.

22.1 Fehlerquellen bei Makros

Aber wo viel Licht ist, ist auch Schatten. Makros bergen auch einige Gefahren in sich, die zu recht eigenartigen Fehlern führen können. Einige können aber mit wenig Aufwand verhindert werden:

Setzen wir in unseren gerade definierten Makroaufruf andere Parameter ein, die noch Operatoren enthalten, so kann es zu Fehlern kommen:

```
erg = MAX(i1 | 2,i2);
```

wird wie gewohnt in

```
erg = ((i1 | 2>i2)? i1 | 2 : i2);
```

umgewandelt. Und hier steckt der Fehler. Haben Sie ihn entdeckt? Wenn Sie sich nicht schon öfter mit den Prioritäten der Operatoren befaßt haben, dürfte das wohl schwerlich möglich sein. Der ">"-Vergleichsoperator hat nämlich eine höhere Priorität als "|". Dies bedeutet, daß zuerst geprüft wird, ob i2 kleiner als 2 ist. Das Ergebnis dieses logischen Vergleiches (0, wenn falsch, 1 wenn wahr) wird dann mit i1 bitweise durch "oder" verknüpft. Mit einer Maximumberechnung hat das aber nicht mehr viel gemeinsam. Bei den Grundrechenarten kommt es nicht zu diesem Verhalten, da diese wiederum eine höhere Priorität als der Vergleichsoperator besitzen. (Eine Tabelle mit allen Prioritäten finden Sie im Anhang). Einfache Abhilfe bildet die

Klammerung aller in dem Makro vorkommenden Parameter, also lieber die Definition

```
#define MAX(a,b) (((a)>(b))? (a):(b))
```

verwenden.

Gegen Seiteneffekte, das sind solche Konstruktionen, bei denen im Funktionsaufruf auch noch ein bißchen rumgerechnet und Werte verändert werden, ist man allerdings machtlos. Ein einfaches Beispiel zeigt das folgende kurze Programm, das das Quadrat der Zahlen 0 bis 10 errechnen soll.

```
#define QUADRAT(x) ((x)*(x))

main()
{
  /* Fehlerhafte Benutzung eines Makros */
  int i = 0;
  while(i <= 10)
    {
      printf("Das Quadrat von %d ", i);
      printf("ist %d\n", QUADRAT(i++));
    }
}
```

Die Ausgabe dieses Programm lautet:

```
Das Quadrat von 0 ist 0
Das Quadrat von 2 ist 6
Das Quadrat von 4 ist 20
Das Quadrat von 6 ist 42
Das Quadrat von 8 ist 72
Das Quadrat von 10 ist 110
```

Wo liegt hier nun der Fehler? Den kann man nur finden, wenn man sich ansieht, was der Präprozessor dem Compiler hinterläßt. Wichtig ist hier natürlich die Zeile mit dem Makro, darüber reden wir ja. Aus

```
printf("ist %d\n",QUADRAT(i++));
```

wird

```
printf("ist %d\n",((i++)*(i++)));
```

Jetzt müßten Sie die Macke erkennen können. Wenn das Quadrat beispielsweise von 2 (i=2) berechnet werden soll, so wird in Wirklichkeit folgender Ausdruck errechnet und der printf-Funktion übergeben.

```
2 * 3
```

Vor der Multiplikation wird bereits die Variable inkrementiert, so daß der zweite Multiplikator falsch ist.

Sicherlich werden Sie sich gewundert haben, warum das Programm zwei printf-Aufrufe benutzt, anstatt kurz und bündig mit dem einen nötigen Vorlieb zu nehmen. Dies liegt daran, das auch hier eine Fehlerquelle zu finden ist, die separat betrachtet wird. Sonst würden wir auch schnell den Überblick vor allen möglichen Nebenwirkungen verlieren. Um den bei der printf-Funktion auftretenden Fehler zu zeigen, probieren wir es mit der gleichen Funktion, diesmal aber ohne Makro.

```
main()
{
  /* Fehlerhafte Benutzung eines Makros */
  int i = 0;
  while(i <= 10)
    printf("Das Quadrat von %d ist %d\n", i, i * i++);
}
```

Als Ergebnis erhalten wir zwar Quadratzahlen, leider gehören sie aber nicht zu den ihnen zugeordneten Werten. Für 3 wird beispielsweise 4 als Quadrat angegeben. Die Quadratzahl ist immer nur für die vorherige Zahl richtig berechnet. Dies liegt daran, daß die Parameter der Funktionen auf einem Zwischenspeicher, dem sogenannten Stack, gespeichert werden und dort von der Funktion erwartet werden. Leider erfolgt die Ablage dieser Werte in umgekehrter Reihenfolge, d.h. zuerst wird "i * i++" abgespeichert (dabei wird natürlich i inkrementiert) und dann

kommt erst der erste Parameter "i" an die Reihe, und der hat jetzt schon den falschen Wert. Was zeigen uns diese Beispiele?

C bietet viele Möglichkeiten, kurze und effiziente Programme zu schreiben. Aber hier besteht auch die Gefahr, zuviel auf einmal machen zu wollen. Die sogenannten Seiteneffekte sollten bei Funktionsaufrufen und Makros entfallen, wenn man nicht hundertprozentig deren Auswirkungen auf alle Variablen und Parameter kennt. Auch sollte die Tabelle der Prioritäten einzelner Operatoren stets zur Hand sein.

22.2 Makros für die Bibliothek

Makros, die häufig verwendet werden sollen, sind in einer Bibliothek am besten aufgehoben, da sie leicht mit "#include" in alle Dateien eingebunden werden können. Zu dieser Gruppe gehören diverse Umwandlungsausdrücke für Buchstaben, z.B. um zu prüfen, ob es sich bei einem zu untersuchenden Zeichen um einen Groß- oder Kleinbuchstaben handelt. Die nachfolgenden Defines haben folgende Aufgaben:

Wandelt Großbuchstaben in Kleinbuchstaben um:

```
#define to_lower(c) ((c)+32)
```

Wandelt Kleinbuchstaben in Großbuchstaben um:

```
#define to_upper(c) ((c)-32)
```

Test auf Buchstabe (ja = 1, nein = 0):

```
#define isalpha(c) ((c)>='A' && (c)<='Z' || (c)>='a' && (c)<='z')
```

Test auf Großbuchstabe (ja = 1, nein = 0):

```
#define isupper(c) ((c)>='A' && (c)<='Z')
```

Test auf Kleinbuchstabe (ja = 1, nein = 0):

```
#define islower(c) ((c)>='a' && (c)<='z')
```

Test auf Ziffer (ja = 1, nein = 0):

```
#define isdigit(c) ((c)>='0' && (c)<='9')
```

Test auf alphanumerisches Zeichen (Buchstabe oder Ziffer) (ja = 1, nein = 0):

```
#define isalnum(c) (isalpha(c) || isdigit(c))
```

Test auf Leerzeichen (dazu gehören auch Tabulator, Zeilenvorschub, Carriage return und Seitenvorschub) (ja = 1, nein = 0):

```
#define isspace(c) ((c)==' '||(c)=='\t'||(c)=='\r'||(c)=='\n'||(c)
==='\f')
```

Test auf Sonderzeichen (ja = 1, nein = 0):

```
#define ispunct(c) ((c)>=' ' && !isalnum(c))
```

Test, ob druckbares Zeichen (ja = 1, nein = 0):

```
#define isprint(c) ((c)>=040 && (c)<=0176)
```

Test auf Steuerzeichen (ja = 1, nein = 0):

```
#define iscntrl(c) ((c)>=0 && ((c)==0177 || (c)<' '))
```

Test, ob ASCII-Zeichen (ja = 1, nein = 0):

```
#define isascii(c) ((c)>=0 && (c)<0200)
```

Diese Defines sollten Sie, wenn Sie sie untersucht und verstanden haben (dürfte keine Schwierigkeiten machen!), in eine eigene Datei namens CTYPE.H schreiben, falls Sie so eine Datei

nicht schon in irgendeiner Subdirectory zur Verfügung haben. Sollte also in einem der folgenden Programme die Zeile

```
#include <ctype.h>
```

vorkommen, wissen Sie, was sich dort zu befinden hat. Noch eine Bemerkung zu den Tests auf Buchstaben: Es werden dort lediglich die 26 Zeichen des Alphabets beachtet. Internationale Sonderzeichen werden nicht als Buchstaben betrachtet, so daß obige Definitionen auf Umlaute nicht anwendbar sind. Vielleicht erweitern Sie diese ja in der geeigneten Weise. Wenn wir gerade bei Anregungen zum Programmieren sind, wir wär's, wenn Sie die strcmp-Funktion mit den neu geschaffenen Defines umschreiben. Keine einzige Funktion, die Ihnen vom Compiler im Zusammenhang mit Vergleichen bereitgestellt wird, beachtet deutsche Umlaute. Einige im Lieferumfang der C-Compiler enthaltenen String-Vergleichsfunktionen sind hier aufgelistet:

```
char *s, *t;
int n, vergl;

vergl = strcmp(s, t);
vergl = strncmp(s, t, n);
vergl = stricmp(s, t);
vergl = strnicmp(s, t, n);
```

Die erste Funktion ist mit unserer selbstgestrickten Routine identisch. Sie vergleicht zwei Strings und liefert das Ergebnis des Vergleiches zurück. Bei der zweiten Funktion strncmp gibt der dritte Wert an, bis zu welcher Stelle der Vergleich durchgeführt werden soll. Sie könnnen dadurch den Vergleich auf "n" Zeichen beschränken, beispielsweise auf die ersten 4 Elemente. Die Funktionen, die noch ein "i" im Namen mitführen, machen keinen Unterschied zwischen Groß- und Kleinschreibung. Dadurch liefert ein Vergleich der beiden Strings "aBcDEf" und "ABcdeF" mit diesen Funktionen den Wert Null, beide Strings sind "gleich".

23. Kontakt mit der Außenwelt

Nachdem unsere Programme lediglich vor sich hin gearbeitet und Ihre Daten vielleicht mal auf dem Monitor ausgegeben haben, sollen sie jetzt auch mit anderen Geräten Verbindung aufnehmen. Die Eingabe von Informationen war bisher auf die Tastatur während des Programmlaufes begrenzt. Die einfachste Möglichkeit, die der AMIGA bietet, dem Programm bereits beim Starten Daten mit auf den Weg zu geben, ist die Übergabe mit Hilfe der Kommandozeile.

23.1 Datenübergabe mittels Kommandozeile

Sie rufen mit dem CLI alle Programme durch Eingabe des Dateinamens und nachfolgend eventuelle Parameter auf. Nehmen wir den Aufruf des Editors ED.

```
ED datei.c SIZE 50000
```

Diese gesamte Zeile kann dem aufgerufenen Programm zur Verfügung gestellt werden, wenngleich auch nicht in dieser Form. Sie wird vorher noch ein wenig vom Betriebssystem für uns aufbereitet.

Nun stellt sich die Frage, wie die Daten an das Programm übergeben werden. Dies wird über die main-Funktion geregelt, die in diesem Fall zwei Parameter erhält. Bisher war jeder Aufruf folgender Natur:

```
main()
{
  ...
}
```

Nun erhält sie vom aufrufenden Programm, eventuell vom CLI oder einer MAKE-Datei, zwei Werte Übergeben. Zum einen erhält man die Anzahl der Informationen und zum anderen einen Zeiger auf - jetzt nicht nervös werden - char-Zeiger. Klingt zuerst etwas kompliziert, wenn wir uns aber anschauen, wie unsere Eingabezeile unserem Programm übergeben wird, dürfte

der Frust weichen. Zuerst die neue Formulierung von main mit den zwei Parametern:

```
main(argc, argv)
int argc;
char *argv[];
{
  ...
}
```

Der Name unseres fiktiven Programms lautet "prg". Betrachten wir eine mögliche Eingabe wie

```
prg Text1 parameter 3 -pi
```

Nach Aufruf unseres Programmes steht in der Variablen "argc" die Anzahl der Parameter, nämlich 5. Sind Sie erstaunt? Warum wird 5 übergeben, wenn nur 4 Parameter vorliegen? Dies liegt daran, daß der beim Aufruf benutzte Programmname ebenfalls hinzugerechnet wird. Der Name "argc" ist übrigens beliebig, da es ja eine auto-Variable der main-Funktion ist. Die beiden Namen haben sich aber eingebürgert und werden fast immer für diesen Zweck verwendet. Die Namen stellen die Abkürzung für "ARGument Count" und "ARGument Vektor" dar.

Jeder Parameter der Eingabezeile wird durch ein Leerzeichen oder einen Tabulator getrennt, die sich aber nicht mehr innerhalb der dann uns übergebenen Daten befinden. Nach obigem Aufruf zeigen somit die Zeiger von *argv[] auf:

```
argv[0] "prg"
argv[1] "Text1"
argv[2] "Parameter"
argv[3] "3"
argv[4] "-pi"
```

Wenden wir uns nun den für uns bestimmten Daten zu. Diese können mit dem folgenden kurzen Programm ausgedruckt werden.

```
main(argc, argv)
int argc;
char *argv[];
{
  while(--argc >= 0)
    puts(*argv++);
}
```

Wofür können wir das nun einsetzen? Vielleicht greifen wir nochmals unser kleines Arithmetikprogramm auf, diesmal erfolgt aber die Eingabe über die Kommandozeile, etwa so:

```
rechne 123.5 * 4711
```

Das Programm soll natürlich nicht ausufern, Sie müssen es schließlich auch noch abtippen. Wir begnügen uns mit dem Ausrechnen einer einfachen Aufgabe, die aus zwei Zahlen und einem Operator besteht. Auf ganze Terme, mit Punkt- vor Strichrechnung und Klammerregeln, können wir nicht eingehen.

```
extern double atof();  /* Deklaration */

int fehler = 0;

main(argc, argv)
int argc;
char *argv[];
{
  double erg, wert();

  if(argc != 4)
    printf("\nFalsche Eingabe\nAufruf: rechne ZAHL1 # ZAHL2\n");
  else
    {
      erg = wert(argv[1], argv[2], argv[3]);
      if(!fehler)
        printf("\n%s %s %s = %.9lf\n", argv[1], argv[2], argv[3], erg);
    }
}
```

```
double wert(zahl1, op, zahl2)
char *zahl1, *op, *zahl2;
{
  double z1 = atof(zahl1);
  double z2 = atof(zahl2);

  switch(*op)        /* Nur das erste Zeichen */
    {
      case '/':
        return(z1 / z2);
      case '*':
        return(z1 * z2);
      case '-':
        return(z1 - z2);
      case '+':
        return(z1 + z2);
      default:
        printf("\n%cUnbekannter Operator >%s<\n", 7, op);
        fehler = 1;
        return(0.0);
    }
}
```

Wir werden später noch ein paar andere Gelegenheiten finden, unser Wissen über die Kommandozeile in weitere Programme einfließen zu lassen.

23.2 Ein-/Ausgabe

Für viele Programme ist es unerläßlich, Daten dauerhaft zu speichern, sei es eine Adreßdatei oder eine Textverarbeitung. Dazu werden Routinen benötigt, die die Ein- und Ausgaben auf externe Geräte wie Drucker, Diskettenstation, RS 232-Schnittstelle oder Festplatte steuern können. Vom Betriebssystem stehen dem Programmierer diverse Funktionen zu diesem Zweck zur Verfügung. Diese Routinen werden in zwei Gruppen getrennt, die gepufferte und ungepufferte Ein-/Ausgabe.

23.2.1 Gepufferte Ein-/Ausgabe

Die Art des Datentransfers erledigt für uns eine Menge Arbeit, die auf den ersten Blick gar nicht zum Vorschein kommt. Alle Daten, die z.B. auf die Diskette übertragen werden sollen, werden zuerst in einem Puffer (Buffer, engl.) zwischengespeichert. Wenn dieser komplett gefüllt ist, werden die Daten wirklich auf Diskette geschrieben. Nun fragt man sich, wozu dieser Umstand?

Ein Diskettenlaufwerk kann Informationen wesentlich langsamer schreiben und lesen, als es dem Computer genehm wäre. Dies liegt daran, daß das Laufwerk nun mal durch Mechanik gesteuert wird. Vor dem Schreiben irgendwelcher Daten muß das Laufwerk erst einmal den Schreib-/Lesekopf auf die Spur bringen, die die Daten aufnehmen soll. Dort muß er solange verharren, bis sich die unter ihm drehende Diskette an der richtigen Position befindet. Dann erst können die Daten geschrieben werden. Für den Computer (genauer seine Zentraleinheit) sind diese für den Menschen recht kurzen Zeitspannen aber gigantisch. Würde jedes einzelne Zeichen mit diesem Verfahren übertragen, würde der Rechner mehr Zeit mit Warten verbringen, um danach ein einziges Zeichen zur Floppystation zu senden, als mit allen anderen zu bewältigenden Arbeiten. Deshalb werden kleinere Datenmengen vor dem Übertragen in einem Speicherbereich des Rechners gesammelt, der, wenn er voll ist, komplett an das Laufwerk geschickt wird. Dadurch verringert sich die Zeit, die der Rechner auf langsame Peripherie warten muß, enorm. Sobald die Diskettenstation das erste Zeichen an der entsprechenden Stelle geschrieben hat, kann es die anderen Daten meist gleich dahinter plazieren, und so in einem Rutsch den gesamten Puffer speichern. Hierdurch verringert sich die Zahl der Diskettenzugriffe, was wiederum zu einer Beschleunigung des gesamten Programmablaufs führt. Dasselbe Prinzip wird ebenfalls beim Lesen von Daten angewendet. Sollten Sie also viele kleinere Datenpakete versenden wollen, so bietet sich diese Methode an.

Die Funktionen, die diesen Job für uns übernehmen, lauten getc und putc, die genau wie ihre Verwandten getchar und putchar ein Zeichen einlesen oder ausgeben.

Diese Funktionen sind, wie Sie wissen sollten, nicht Teil der Sprache C, den Sprachumfang haben wir ja bereits abgehandelt. Daher finden Sie solche Routinen in der Regel in einer Bibliothek oder, und das ist hier der Fall, als Define in einer Header-Datei (".h"). Die Definition der getc und putc finden Sie beispielsweise ebenso in der "stdio.h"-Datei, wie die altbekannten putchar und getchar.

```
#define getc(p) (--(p)->_rcnt>=0? *(p)->_ptr++:_filbf(p))
#define getchar() getc(stdin)
#define putc(c,p) (--(p)->_wcnt>=0?((int)(*(p)->_ptr++=(c))):_flsbf((c),
p))
#define putchar(c) putc(c,stdout)
```

Undurchschaubarer und komplizierter geht es kaum noch. Wir brauchen uns mit diesen Ausdrücken auch nicht weiter zu beschäftigen, lediglich die Definitionen von putchar und getchar sind bemerkenswert. Beide sind auf getc und putc zurückgeführt, so daß sie eine Spezialversion dieser beiden Funktionen darstellen.

Bevor wir mit diesen Funktionen die ersten Zeichen transportieren können, müssen wir erst einmal einen Kanal öffnen. Ein Kanal ist nichts anderes, als eine Datenleitung zu einem bestimmten Gerät. Es wird für uns natürlich keine Leitung im Sinne eines Kabels freigehalten, aber dem System ist danach bekannt, wohin die Informationen gesendet werden sollen. Mit Hilfe einer speziellen Kennung, die wir beim Öffnen eines Kanals vom Betriebssystem erhalten, können wir jederzeit das festgelegte Gerät ansteuern. Man kann aber nicht nur Geräte ansteuern, sondern auch einzelne Dateien, z.B. auf Diskette. So können mehrere Dateien auf ein und demselben Laufwerk angesprochen werden, ohne daß sich die Daten in die Quere kommen. Dafür haben wir den FILE-Pointer, die schon angesprochene Kennung eines Kanals.

Da wir einen Puffer für unsere Ein-/Ausgabe benutzen, müssen wir dem Rechner auch mitteilen, wo er die Daten zwischenspeichern soll und wie groß der dafür zu reservierende Platz ist. Auch diese Arbeit wird uns durch eine Strukturdefinition in "stdio.h" abgenommen. Mit dem Namen "FILE" (großgeschrieben!) ist dort eine Struktur vorzufinden, die alle nötigen Teile einer gepufferten Ein-/Ausgabe enthält. Ein einfaches Beispiel, das eine Datei beschreibt und danach wieder ausliest, zeigt das folgende Listing.

```
#include <stdio.h>


main()
{
  FILE *eingabe, *ausgabe, *fopen();
  int c;
  char dateiname[81], text[200];

  printf("Bitte Dateinamen eingeben!\n");
  scanf("%80s", dateiname);
  printf("Jetzt können Sie ein (langes) Wort eintippen\n");
  scanf("%200s", text);

  ausgabe = fopen(dateiname, "w");
  printf("Filehandle %d\n", ausgabe);
  fprintf(ausgabe, "%s", text);
  fclose(ausgabe);

  eingabe = fopen(dateiname, "r");
  printf("Filehandle %d\n", eingabe);
  fscanf(eingabe, "%200s", text);
  fclose(eingabe);
  printf("Der Text >%s<\n", text);
}
```

Die Funktion fopen öffnet den Kanal und gibt uns den Filepointer zurück, der als Ausweis für alle weiteren Zugriffe auf diese Datei dient. fopen muß, da es etwas anderes als Integer liefert, auch als Funktion, die einen FILE-Pointer liefert, deklariert werden. Diese Routine verlangt von uns zwei Parameter,

zum einen den Namen der Datei, zum anderen die Wahl des Zugriffes. Den Namen können Sie zu Beginn selbst eingeben. Der Modus, der als zweiter Parameter der fopen-Funktion darüber Auskunft gibt, was mit der Datei gemacht werden soll, kann drei verschiedene Werte annehmen:

"r" (read) öffnet eine Datei zum Lesen.
"w" (write) öffnet eine Datei zum Beschreiben.
"a" (append) öffnet eine Datei zum Anhängen weiterer Daten.

Wird eine Datei zum Lesen geöffnet ("r"), so können eben nur Daten gelesen, aber nicht geschrieben werden. Bei dem Modus "w" und "a" ist dies genau umgekehrt, wobei durch Append die Daten an eine bereits bestehende Datei angehängt werden. Mit dem normalen Beschreiben "w" wird stets bei Dateibeginn angefangen, und bereits bestehende Informationen werden überschrieben. So kann es also leicht zu Datenverlusten kommen. Im obigen Beispiel wird bei jedem Programmstart die angegebene Datei überschrieben, so daß die zuletzt gespeicherten Daten verloren sind.

Nach dem Öffnen steht die Datei zum Beschreiben bereit. Damit Sie mal sehen, wie so ein File-Handle aussieht, wird es auf dem Bildschirm angezeigt. Die per Tastatur eingegebenen Zeichen werden mittels scanf dem String "text" zugewiesen. Über die Funktion fprintf können wir Daten in eine Datei schreiben. Sie ist fast identisch mit der bekannten printf-Routine, unterscheidet sich aber durch den ersten Parameter. Ihr muß vor dem Kommandostring erst noch das File-Handle übergeben werden, damit die Informationen auch in die richtige Datei gelangen. Nach dem Beschreiben wird die Datei wieder geschlossen. Dieser Schritt ist sehr wichtig, da wir ja von der Existenz des Puffers wissen. Dort werden alle Ein- bzw. Ausgaben zwischengespeichert, bis der Puffer gefüllt ist. Deshalb werden einige unserer eingetippten Daten bestimmt noch in diesem Speicherbereich gelagert sein. Würden wir im guten Glauben, alles sei auf Diskette abgespeichert, den Rechner ausschalten, wäre alles, was sich noch im Puffer befände, verloren. Deshalb wird durch den fclose-Aufruf nicht nur der Kanal geschlossen, sondern zuvor

auch noch der restliche Inhalt des Puffers in die geöffnete Datei geschrieben.

Nun öffnen wird das File erneut, diesmal aber zum Lesen, was wir durch Angabe von "r" signalisieren. Da wir zur Dateneingabe von der Tastatur stets scanf benutzten, verwenden wir hier ebenfalls den Verwandten fscanf. Auch hier müssen zuerst der FILE-Pointer und dann erst die gewohnten Parameter der scanf-Routine unserer Lesefunktion übermittelt werden.

Danach folgt das korrekte Schließen des Files. Auch wenn dieser Befehl ausgelassen wird, ist nicht unbedingt mit Datenverlust zu rechnen. Trotzdem sollte man sich an die Regel halten, ein geöffnetes File nach Gebrauch sofort wieder zu schließen, nicht nur der Ordnung halber, sondern einfach deswegen, weil jeder Rechner nur eine bestimmte Anzahl gleichzeitig geöffneter Kanäle verkraften kann. Werden laufend weitere Dateien geöffnet, ist irgendwann kein Kanal mehr frei. Sollte ein Fehler aufgetreten sein, daß das Betriebssystem Ihnen keinen Kanal zur Verfügung stellen kann, so erhalten Sie als FILE-Pointer Null zurück. Dies ist zum Beispiel der Fall, wenn keine Kanäle mehr frei sind oder die zu öffnende Datei, die Sie lesen möchten, überhaupt nicht existiert.

Schreiben wir zur Übung ein kleines Kopierprogramm. Es soll mit Parametern aufgerufen werden können, muß also von main Argumente entgegennehmen. Nun gilt es aber eine Schwierigkeit zu umschiffen. Wir wissen ja nicht im Vorhinein, um welche Art es sich bei den zu übertragenden Daten handelt. Wir können die fscanf nicht verwenden, da dort angegeben wird, ob man es mit Strings oder Zahlen zu tun hat. Es bleibt uns nur das zeichenweise Einlesen. Dazu könnte man zwar wieder die scanf-Routine mit der Formatanweisung "%c" benutzen, dazu eignet sich aber eine andere Funktion viel besser: fgetc. Sie liest aus einer Eingabedatei lediglich ein Zeichen und ist dadurch viel schneller als die universelle fscanf-Funktion. Umgekehrt geben wir ein einzelnes Zeichen nicht mit fprintf, sondern mittels fputc aus.

Nach dem Öffnen beider Dateien lesen wir ein Zeichen ein und geben es sofort wieder aus, bis... Tja, wir wissen ja gar nicht, wann alle Daten kopiert sind. Wie sollen wir erkennen, wann das letzte Zeichen bearbeitet wurde. Gott sei Dank ist dieses Problem schon gelöst worden. fgetc liefert uns ein besonderes Zeichen, wenn keine Informationen mehr bereit stehen: End Of File (Ende der Datei) oder kurz EOF. In dem Includefile "stdio.h" ist dieser Text bereits als Define festgelegt, so daß wir lediglich das eingehende Zeichen mit "EOF" vergleichen müssen.

EOF ist dort als -1 festgelegt. Das hat für uns, auch wenn man es nicht auf den ersten Blick sieht, Konsequenzen. Gültige Daten, bei fgetc sind dies Zeichen aller Art, besitzen Codes zwischen 0 und 255. Wenn aber auch -1 auftauchen kann, dürfen wir keine char-Variable zur Datenaufnahme wählen. Entweder werden negative Zahlen "übersehen", oder es gehen Daten verloren. Deshalb verwendet man int-Variablen, auch wenn darin nur ein char-Element abgespeichert wird.

```
#include <stdio.h>


main(argc, argv)
int argc;
char *argv[];
{
  long copy(); /* Wen es interessiert */

  if(argc != 3)
    {
      printf("Fehlerhafte Argumente!\n");
      printf("Vondatei Nachdatei\n");
    }
  else
    copy(argv[1], argv[2]);
}
```

```
copy(vondat, nachdat)  /* Kopierroutine */
char *vondat, *nachdat;
{
  FILE *eingabe, *ausgabe, *fopen();
  register long anzahl = 0;
  register int c;

  if(!(eingabe = fopen(vondat, "rb")))  /* als Binaerdatei oeffnen */
    {
      printf("%s kann nicht geoffnet werden!\n", vondat);
      return 0L;
    }
  if(!(ausgabe = fopen(nachdat, "wb")))
    {
      printf("%s kann nicht geöffnet werden!\n", nachdat);
      fclose(eingabe);  /* Der war ja in Ordnung */
      return 0L;
    }

  while( (c = fgetc(eingabe)) != EOF)
    {
      fputc( c, ausgabe);
      anzahl++;
    }

  fclose(eingabe);
  fclose(ausgabe);
  printf("\n%ld Bytes kopiert!\n", anzahl);
  return(anzahl);
}
```

Dieses Kopierprogramm wird folgendermaßen aufgerufen:

```
kopier [d:][\path]name1[.ext] [d.][\path]name2.[ext]
```

Alles, was in eckigen Klammern geschrieben steht, ist optional, kann also weggelassen werden. Es müssen lediglich zwei Dateinamen angegeben werden. Sollte ein Fehler beim Öffnen einer der beiden Dateien aufgetreten sein, so erscheint eine Fehlermeldung. Das gleiche passiert, wenn zu wenig oder zu viele Parameter übergeben werden (Es müssen zwei sein!).

Dieses Programm ist selbstverständlich nicht zum täglichen Kopiergebrauch geeignet, da es relativ langsam ist und man ohnehin mit dem DOS-Kommando "copy" ein leistungsfähiges Programm zur Verfügung hat. Als Anschauungsbeispiel ist es aber hervorragend geeignet. Beschränken wir uns auf das Wesentliche. Die Dateien werden als Binärdateien durch "rb" bzw. "wb" geöffnet. (Benutzer des Aztek-Compilers müssen das "b" weglassen. Die Datei wird immer als Binärdatei eröffnet.) Dadurch wird verhindert, daß irgendwelche Umwandlungen vorgenommen werden, wie dies z.B. bei Textdateien sinnvoll ist. So werden automatisch beim Lesen alle '\r'-Zeichen (Carriage Return) gelöscht und Zeichen mit dem Code 26 (CTRL-Z) in EOF umgewandelt. Andererseits wird beim Schreiben aus einem Linefeed '\n' die Zeichenkombination '\r' und '\n'. Durch Öffnen einer Datei mit dem Anhängsel "b" erhalten wir alle Zeichen so, wie sie auch in der Datei abgespeichert sind. Umgekehrt werden auch nur die Daten in ein File geschrieben, die von uns abgeschickt wurden.

23.3 Weitere gepufferte Ein-/Ausgabefunktionen

Neben fgetc, fputc, fscanf und fprintf gibt es noch einige recht wichtige Funktionen, die den internen Puffer ausnutzen. Dazu gehören fread und fwrite. Diese Routinen transportieren nicht nur ein einzelnes Byte, sondern eine beliebige Anzahl. Daher sind bei fread und fwrite zwei Parameter mehr nötig als bei den bisherigen getc und putc. Zum einen ist dies ein Bereich, der als Buffer dient, zum anderen kommt auch noch die Größe der zu übertragenden Einheiten hinzu. Das bedarf einiger Erläuterungen. Der Buffer wurde bei den zuvor verwendeten Funktionen stets in der FILE-Struktur festgelegt. Da dort nur kleine Datenmengen transportiert werden, braucht der Buffer nicht besonders groß zu sein (512 Byte). Da Sie nun selbst bestimmen können, welche Datenmenge übertragen werden soll, kann es schnell zu einem zu kleinen Buffer kommen. Daher müssen Sie den Speicherbereich selbst definieren, dadurch die Maximalgröße des Datentransfers festlegen, und den Funktionen übergeben.

Nun zur Datengröße, die ebenfalls angegeben werden muß. Bei getc und putc kann nur immer ein Zeichen (1 Byte) übertragen werden, somit erübrigt sich eine zusätzliche Angabe der Größe eines char-Objektes. Sollen z.B. nicht Zeichen, sondern long-Werte gespeichert werden, so müssen bei 10 solchen Zahlen ja nicht 10, sondern 40 Byte übertragen werden. Die Größe dieser Objekte, in diesem Falle 4 (Bytes), ist der zweite zu übergebende Parameter. Es empfiehlt sich, wenn man den Speicherplatz eines Datentypes nicht ganz sicher weiß, den sizeof-Operator zu verwenden, der diesen Wert liefert.

Außer diesen beiden Parametern sind noch die Anzahl der zu übertragenden Einheiten (char, int, Struktur,...) und natürlich wieder der FILE-Pointer zu übergeben. Die Puffergröße, die ja bei der Definition festgelegt wird, sollte auf keinen Fall zu klein gewählt werden. Ein Aufruf dieser Funktion sieht folgendermaßen aus:

```
Datentyp Buffer[Elemente]; /* Definition des Buffers */

fread(Buffer, sizeof(Datentyp), Elemente, file_ptr);
```

Wie das Beispiel zeigt, ist der erste Parameter der Puffer, aus dem die Daten gelesen werden. Der Buffer sollte vom gleichen Typ wie die zu übertragenden Einheiten sein, um die Übersicht zu behalten. Als zweiter Wert ist die Größe einer Einheit anzugeben. Diese Einheit ist ein Datenpaket, das als untrennbare Einheit übertragen werden kann. Wenn wir long-Variablen übertragen, so ist es sinnvoll, 4 Byte, eben der Speicherbedarf eines solchen Wertes, als Datenblocklänge anzugeben. Nun folgt die Anzahl der Datenblöcke, die die zuvor angegebene Datenlänge besitzen. Hier muß die Anzahl, nicht der gesamte Speicherbedarf übermittelt werden. Sind 100 double-Variablen zu speichern, so wird an der Stelle "Elemente" 100 plaziert. Zum Schluß fügen wir noch unseren von fopen erhaltenen Filepointer hinzu.

Ob wir Daten lesen oder schreiben, die Anordnung und Art der Parameter ist stets dieselbe. So werden Daten mit fwrite beispielsweise durch folgende Sequenz gespeichert:

```
int tabelle[876], groesse = 876;
FILE *ausg_ptr;

fwrite(tabelle, sizeof(double), groesse, ausg_ptr);
```

Wollen Sie den Wert "groesse" nicht explizit angeben, sei es durch eine dafür vorgesehene Variable, oder durch ein Define, kann man auch wieder den sizeof-Operator zu Rate ziehen. Der Ausdruck

```
sizeof(tabelle)/sizeof(int)
```

wäre das Äquivalent zur Variablen "groesse". Die Anzahl der vollständig übertragenen Datenpakete liefern uns die Funktionen als Rückgabewert. So können wir beispielsweise kontrollieren, ob wirklich alle Daten abgespeichert werden konnten. Beim Kopieren wäre es auch möglich, solange Daten einzulesen, bis die Anzahl der angeforderten von der Zahl der gelieferten Daten abweicht. Sollte man als Resultat von fread eine Null erhalten, hat man gerade die letzten Daten gelesen. Ein kleines Programmbeispiel:

```
#include <stdio.h>

#define ANZ_DATEN (sizeof(daten)/sizeof(long))


long daten[] =
{
  4711, 815, 1024, 1, 31415926, 0, -13, 10, 0xFFFF, 065432
};
```

```
main()
{
  FILE *eingabe, *ausgabe, *fopen();
  int i;
  long test[ANZ_DATEN];
  char dateiname[81];

  printf("Bitte Dateinamen eingeben!\n");
  scanf("%80s", dateiname);
  printf("Datengröße %d, Elemente %d\n", sizeof(daten), ANZ_DATEN);
  ausgabe = fopen(dateiname, "wb"); /* Auf jeden Fall BINAER! */
  fwrite(daten, sizeof(long), ANZ_DATEN, ausgabe);
  fclose(ausgabe);

  printf("Daten einlesen!\n");
  eingabe = fopen(dateiname, "rb");
  printf("%d Elemente eingelesen\n",
    fread(test, sizeof(long), ANZ_DATEN, eingabe));
  fclose(eingabe);
  for(i = 0; i < ANZ_DATEN; i++)
    printf("%ld\t", test[i]);
  printf("\nFertig!\n");
}
```

Dieses Programm schreibt, nachdem Sie einen Namen für die Datei eingegeben haben, ein long-Array in das File. Dabei werden einige Informationen wie Größe des Arrays und Anzahl der Elemente auf dem Bildschirm angezeigt. Durch das Define "ANZ_DATEN", das nicht anderes darstellt als die obige Verallgemeinerung durch sizeof, werden alle Daten schließlich auf Diskette abgespeichert. Nach dem Schließen dieser Datei verwenden wir ein anderes Array, um die einzulesenden Informationen aufzunehmen. Am Schluß werden alle transportierten Daten auf dem Monitor angezeigt.

Sehr wichtig ist, daß Sie auf jeden Fall beim Öffnen der Datei auf das "b" achten. Das File muß als Binärdatei (nur beim Lattice) geöffnet werden da ansonsten durch die interne Umwandlung völlig falsche Werte in den Variablen vorliegen. Wenn Sie fwrite und fread verwenden, müssen Sie die Datei auch als Binärfile öffnen.

23.4 Ungepufferte Ein-/Ausgabefunktionen

Neben den gerade ausprobierten gepufferten Funktionen existieren auch noch Routinen, die keinen Buffer benötigen. Die zu übertragenen Daten brauchen nicht erst in einen Puffer transportiert zu werden, sondern können gleich in den Variablen untergebracht werden. Dadurch verschwindet der gesamte Aufwand, der durch Puffer und interne Zeiger verursacht wird. Ein FILE-Pointer, über den das Betriebssystem auf den Buffer zugreifen kann, ist auch nicht mehr nötig. Lediglich eine Kanalnummer, die beim Öffnen vergeben wird, muß als Identifizierung benutzt werden. Die Kanalnummer wird in eine Integervariable gespeichert und ersetzt den Filepointer bei allen Aufrufen. Die Funktion zum Öffnen einer Datei ist natürlich auch eine andere. Sie heißt nun open. Ihr wird der Dateiname und ein Integerwert für den Modus übergeben. Im Gegensatz zu fopen, bei der der Modus ein String sein muß, ist bei open der Modus einer der Werte 0, 1, 2 oder 8. Diese Ziffern entsprechen den Strings "r", "w" und "a".

0 öffnet ein File zum Lesen
1 öffnet ein File zum Schreiben
2 öffnet ein File zum Lesen und Schreiben
8 öffnet ein File zum Anfügen

Wer sich nicht mit diesen Zahlen herumschlagen möchte, kann zu diesem Zweck auch Defines benutzen. Die sind in einer Headerdatei namens "fcntl.h" untergebracht und lauten wie folgt:

```
#define O_RDONLY 0
#define O_WRONLY 1
#define O_RDWR 2
#define O_APPEND 8
```

Ob das nun übersichtlicher ist? Na ja, wenn Sie aber diese Defines benutzen, müssen Sie die Datei auch durch "#include <fcntl.h>" in Ihren Source einbinden.

Ein weiterer Unterschied zur fopen-Routine besteht darin, daß die open beim Öffnen einer Datei deren Existenz voraussetzt. Während durch Aufruf von

```
fopen("Dateiname","w")
```

eine Datei stets neu erzeugt wird, egal ob die Datei besteht oder nicht, erfordert die open-Funktion immer den Namen einer existierenden Datei. Soll eine neue Datei erstellt werden, muß die creat-Funktion verwendet werden.

creat gibt als Rückgabewert einen Integer, das sogenannte File-Handle zurück, so daß kein Aufruf von open mehr nötig ist. Sollte ein Fehler auftauchen, so daß die Datei nicht geöffnet werden kann, liefern sowohl creat als auch open den Wert -1. Vor der Verwendung dieses Wertes als File-Handle sollte es deshalb auf -1 überprüft werden, da ansonsten, ich mußte es am eigenen Leib erfahren, der Rechner abstürzt. Lediglich durch eine Tastenkombination (CTRL + A + A) war er wieder dem Dämmerzustand zu entreißen, von den Daten in der RAM-Disk ganz zu schweigen.

Geschrieben wird in eine so geöffnete Datei nicht mit fwrite, sondern mit der dafür nötigen Routine write. Dadurch, daß wir bei dieser Methode keinen Puffer benutzen, können auch nur speziell darauf abgestimmte Ein-/Ausgabefunktionen zum Einsatz kommen. Auch die fread-Funktion wird durch das Pendant read ersetzt. Man kann sich das einfach merken. Gepufferte Funktionen besitzen ein "f" zu Beginn ihres Namens (fopen, fread, fclose), bei ungepufferten fehlt dieses. So lautet die Funktion zum Schließen einer ungepufferten Datei einfach close.

Sehen wir uns folgende abgewandelte Funktion an, die diesmal eine Liste von double-Zahlen abspeichert.

```
#define ANZAHL (sizeof(daten)/sizeof(double))

double daten[] =
{
  1.5, 2.0, 3.14159265, 2.718281828,
};


main()
{
  int handle, dummy = 0, i, wirklich;
  double daten2[ANZAHL];
  char dateiname[81];

  printf("Bitte Dateinamen angeben!\n");
  scanf("%80s", dateiname);

  handle = creat(dateiname, dummy); /* Neu erschaffen */
  if(handle != -1)  /* Alles OK? */
    {
      wirklich = write(handle, daten, sizeof(daten));
      printf("Wunsch %d Bytes, Wirklichkeit %d Bytes\n",
        sizeof(daten), wirklich);
      close(handle);
    }
  else
    printf("Fehler beim Erschaffen von %s\n", dateiname);

  handle = open(dateiname, 0, dummy); /* Lesen */
  if(handle != -1)  /* Alles OK? */
    {
      wirklich = read(handle, daten2, sizeof(daten));
      printf("Wunsch %d Bytes, Wirklichkeit %d Bytes\n",
        sizeof(daten), wirklich);
      close(handle);
    }
  else
    printf("Fehler beim Offnen von %s\n", dateiname);
  for(i=0; i < ANZAHL; i++)
    printf("%.8lf  ", daten2[i]);
  printf("\nDas wars!\n");
}
```

Bei beiden Funktionen, die uns ein Handle für die Datei besorgen, open und creat, taucht eine merkwürdige Variable namens dummy auf. Die Variable stellt einen Wert dar, der zwar völlig überflüssig ist, aber vom Compiler dort gewünscht wird. Soll er seinen Willen haben! Der Wert, den Sie in dummy abspeichern, ist, wie der Variablenname "dummy" (engl. Attrappe) aussagt, ohne jegliche Bedeutung. Bei open stellt der zweite Parameter einen der zuvor genannten Werte für Lesen oder Schreiben dar.

Die Funktionen write und read besitzen einen Parameter weniger als ihre Verwandten fwrite und fread. Außerdem - und darauf sollten Sie höllisch achten - steht hier das File-Handle zu Anfang und nicht wie bei den gepufferten Funktionen am Ende. Die ungepufferten Routinen transportieren die Daten nur noch in Byte-Einheiten, so daß die Angabe der Größe eines Datenpaketes entfällt. So erhalten Sie als Rückgabewert auch die Anzahl der tatsächlich übertragenen Bytes geliefert.

23.5 Direktzugriff

Mit den folgenden Funktionen haben wir nun die Möglichkeit, direkt auf bestimmte Zeichen in einer Datei zuzugreifen. Der Unterschied zum üblichen Lesen besteht darin, daß nicht erst diverse Zeichen vom Dateianfang überlesen werden müssen, um an eine bestimmte Position zu gelangen. In einer Datei mit 1000 Zeichen müßten, damit man an die letzten 10 Zeichen heran kommen kann, 990 Zeichen überlesen werden. Durch den Direktzugriff erteilt man den Befehl, erst ab dem 990. Zeichen mit dem Einlesen zu beginnen. Für solche Aktionen steht ein Dateizeiger zur Verfügung, der stets auf die zuletzt bearbeiteten Daten zeigt. Beim sequentiellen Lesen oder Schreiben, wenn also ein Zeichen nach dem anderen bearbeitet wird, wird dieser Zeiger immer um eins erhöht. Durch die Funktion lseek und fseek kann dieser Zeiger auf jede beliebige Position gesetzt werden. Beide Routinen benötigen hierfür 3 Parameter, wobei lseek die ungepufferte und fseek die gepufferte Version ist. Daher benötigt lseek als ersten Parameter das File-Handle, während fseek einen FILE-Pointer erwartet. Als zweiter Wert folgt die Zahl der Bytes, um die der Dateipointer bewegt werden soll. Positive

Werte bewegen den Zeiger in Richtung Dateiende, negative Werte in Richtung Dateibeginn, wobei dieser Wert als long-Wert übergeben werden muß. Der dritte Parameter, ein Integer, gibt schließlich an, von welcher Position die Bewegung erfolgen soll. 0 setzt den Dateizeiger auf den Dateianfang, 2 auf das Dateiende und 1 als letzte Möglichkeit geht von der aktuellen Position aus. Einige Beispiele:

```
lseek(f_handle, 100L, 0);
```

Der Dateizeiger wird auf Position 100 bewegt, also 100 Zeichen vom Dateianfang entfernt. Er steht jetzt auf dem 101. Byte der Datei. Nach der Anweisung

```
lseek(f_handle, 70L, 1);
```

befindet sich der Dateizeiger auf Position 170, da bei diesem Funktionsaufruf der Wert 1 übergeben wird, der die Berechnung der neuen Zeigerposition vom aktuellen Platz vornimmt. Um den Zeiger nun wieder 20 Zeichen nach vorne (Richtung Dateibeginn) zu verschieben, ist der folgende Befehl notwendig:

```
lseek(f_handle, -20L, 1);
```

Soll die Bearbeitung einer Datei am Dateiende beginnen, so kann der Zeiger einfach mit

```
lseek(f_handle, 0L, 2);
```

auf die letzte Position der Datei gesetzt werden. Daraus ergibt sich zwangsläufig, daß im Modus 2 (Dateiende) nur negative Zahlen oder 0 zugelassen werden, da der Zeiger schon am Dateiende steht. Im Modus 0 hingegen dürfen wiederum nur positive Zahlen oder 0 eingesetzt werden. Wie Sie sehen, können Sie mit diesen Funktionen nur den Dateizeiger hin- und herbewegen, die einzelnen Daten müssen mit den diversen bereits bekannten Funktionen wie read oder write gelesen oder geschrieben werden.

Die Funktion lseek gibt den Wert des Dateipointers nach dem Verschieben zurück, während fseek entweder 0 oder -1 liefert. Bei -1 ist ein Fehler aufgetreten, ansonsten lief alles glatt.

Zwei weitere Routinen, jeweils eine gepufferte und eine ungepufferte Version, können den aktuellen Wert des Dateizeigers erfragen. Da dies ein long-Wert ist, müssen diese Funktionen ftell und tell auch zuvor deklariert werden. Der einzig benötigte Parameter beider Funktionen ist das entsprechende File-Handle.

Die Funktion tell ließe sich aber auch durch den Aufruf

```
lseek(f_handle, 0L, 1);
```

ersetzen, da hierdurch genau der Wert des aktuellen Dateizeigers ermittelt wird.

23.6 Einlesen eines Zeichens

Vielleicht standen Sie schon bei einem eigenen Programm vor der Aufgabe, ein einzelnes Zeichen einzulesen. Wir haben uns diese Information bislang immer mit scanf besorgt. Das hat aber einen kleinen Haken. Wir müssen stets nach dem eingetippten Zeichen noch die RETURN-Taste betätigen. Wollen wir eine Textverarbeitung schreiben, kann man damit natürlich nicht arbeiten. Aber selbst, wenn Sie sich mit einer so bescheidenen Aufgabe wie der Steuerung eines Cursors in alle vier Himmelsrichtungen beschäftigen, werden Sie damit Schwierigkeiten bekommen. Blättert man ein wenig in C-Büchern oder hat man vielleicht schon etwas Vorwissen, so stößt man zwangsläufig auf die Funktion getchar. Die Aufgabe dieser Funktion ist das Abliefern des Codes einer gedrückten Taste, und zwar direkt nachdem diese betätigt wurde. "Das ist genau das, was wir wollen!", werden Sie nun meinen. Aber Pustekuchen, hier unterscheiden sich wieder Theorie und Praxis. Auf allen Rechnern, die in C zu programmieren sind, funktioniert diese Funktion, wie es auch von ihr gefordert wird. Woran liegt es aber, daß beim AMIGA auch bei dieser Funktion auf Betätigung der

RETURN-Taste bestanden wird. Doch bestimmt kein böser Wille der Lattice-Programmierer?!

Das ist gar nicht so einfach zu erklären. Wir müssen uns deshalb erst einmal mit einigen grundlegenden Dingen befassen, der

23.6.1 Standardein- und -ausgabe

Nun was kann man sich darunter vorstellen? Der Standard ist ja der Regelfall, so müssen wir uns nur noch überlegen, was das mit unserer Datenein- und -ausgabe zu tun hat.

In der Regel wird man seine Daten über Tastatur, auch Konsole genannt, in den Rechner eintippen. Die Ausgabe, z.B. Rechenergebnisse oder Meldungen, erscheint dann auf dem Monitor. Damit hätten wir bereits die beiden wichtigen Begriffe verstanden. Sollten wir dem Rechner keine näheren Informationen mit auf den Weg geben, woher er seine Daten bezieht und wohin er sie liefern soll, so wird er stets auf die Standardein-/-ausgabe zurückgreifen. Dies sind Tastatur und Bildschirm. Das gute an dieser Geschichte ist, daß diese Geräte zwar voreingestellt sind, aber durch uns geändert werden können. Es ist kein Problem, dem Rechner mitzuteilen, daß er ab sofort die Informationen, die wir über Tastatur eingeben sollen, von einer anderen Stelle bezieht. Umgekehrt kann der AMIGA auch die Order erhalten, alle Ausgaben nicht mehr auf dem Bildschirm auszugeben, sondern an ein völlig anderes Gerät zu schicken.

Die Standardeingabe ist, wie wir nun wissen, die Tastatur. Da muß allerdings noch etwas spezifiziert werden. Genauer wird das nämlich vom aufrufenden Programm, dem CLI, bestimmt. Die Standardausgabe ist ebenso mit "Bildschirm" noch etwas unzureichend gekennzeichnet. Bei den diversen Fenstern schreiben wir ja nicht kreuz und quer über alles, was nicht niet- und nagelfest ist, sondern alle Texte wandern schön ordentlich in das CLI-Fenster. Dieses Fenster ist für unser Programm die Standardausgabe. Alle Beschränkungen, die für das CLI gelten, müssen logischerweise dann auch für unser Programm zutreffen. Und hier liegt das Geheimnis für die fehlerhafte Funktion

getchar. Das CLI arbeitet ja zeilenorientiert. Erst wenn eine komplette Zeile eingegeben und durch RETURN abgeschlossen wurde, werden die Daten verarbeitet. Sie können prinzipiell jeden Mist und Unsinn eintippen, ohne daß Ihnen der Rechner bereits bei der Eingabe meldet, daß es so nicht geht. Tippen Sie doch mal die Tastenkombination CTRL-G, die den Bildschirm kurz aufblinken läßt. Obwohl man dieses Zeichen (Es hat den Code 7) eingeben kann, ist es jedoch unmöglich, es auf dem Bildschirm darzustellen. Wird dies doch getan (zumindest versucht), so wird die hierfür vorgesehene Funktion, das Bildschirmblinken, ausgeführt. Beim CLI hat man also eine Eingabekonsole vor sich, die nur ganze Zeilen und keine einzelnen Tasten verarbeitet.

Genau diese Vorschrift, nur komplette Zeilen zu benutzen, wird leider auch unserem vom CLI gestarteten Programm mit auf den Weg geschickt. Nachdem wir nun wissen, woran es liegt, daß man jedesmal RETURN drücken muß, sollten wir uns überlegen, wie man Abhilfe schafft. Das erste ist, wir müssen uns von den Vorschriften des CLI befreien. Dazu schafft man sich ein eigenes Fenster (Window), dessen Vorschriften von uns selbst bestimmt werden.

23.7 Ein eigenes Fenster

Das Erzeugen eines ganz "privaten" Windows ist relativ einfach. Wir benutzen dazu den Befehl open, der bereits bei der Dateibehandlung gute Dienste geleistet hat. Im Grunde ist es ja auch nichts anderes, als eine Ausgabe- beziehungsweise eine Eingabedatei zu öffnen, da die Ausgaben in dieses neu geschaffene Fenster geschrieben und die Eingaben nach den Richtlinien unseres Windows von der Tastatur gelesen werden.

23.7.1 Die drei Modi

Damit wir uns aber von den Beschränkungen des CLI lösen können, müssen wir eine andere Art Fenster öffnen. Uns stehen drei Möglichkeiten offen:

"*"
"CON:"
"RAW:"

Die Zeichenketten werden bei open anstelle des Dateinamens und der Laufwerkangabe verwendet. Beim Stern erhalten wir kein neues Window, wir schreiben weiterhin alles ins CLI-Fenster. Auch bei der Eingabe hat sich nichts geändert. Es bringt uns also nicht den geringsten Vorteil, erst eine Datei mit dem Stern zu öffnen, um nachher vor den gleichen Problemen zu stehen. Interessant wird es allerdings mit "CON:", hier bekommen wir unser erstes eigenes Window. Sehen wir uns den open-Aufruf einmal an:

```
open("CON:0/0/200/50/Titelzeile", 0, dummy)
```

Wir benutzen erneut die ungepufferte open-Routine, der aber ein ganz eigenartiger "Pfadname" übergeben wird. Anstelle der Laufwerksangabe benutzt man "CON:" (Console), gefolgt von den Koordinaten des neu zu schaffenden Fensters. Am Schluß gibt man dem Window noch eine Überschrift, die dann in der Titelzeile links oben erscheint. Alle Angaben sind durch den Schrägstrich (Slash) "/" zu trennen. Die anderen Parameter wie 0 (Lesen) und der Dummy-Wert folgen nur dem bekannten Aufbau der open-Routine.

Das Datei-Handle, das wir geliefert bekommen, verwenden wir wie in den vorhergehenden Beispielen bei allen read-Aufrufen. Als Zwischenspeicher verwenden wir vorsichtshalber einen String, der mit 81 Einträgen nicht zu knapp bemessen ist. Vielleicht probieren Sie unsere neue kleine Routine mal aus?!

```
#define c *zeichen
#define ESC 27

main()
{
  int dummy = 0, anz, handle;
  char zeichen[81], zeile[256];

  handle = open("CON:0/0/200/50/Mein Programm", 0, dummy);
  printf("Eröffnet %d handle\n", handle);
  if(handle != -1)
    {
      do
        {
          anz = read(handle, zeichen, 1);
          printf("Zeichen >%c< Code %d\n", c, c);
        } while(c != ESC);
      close(handle);
    }
}
```

Sie sollten den Test auf ein erfolgreiches Öffnen der Datei nicht vergessen, da Ihnen sonst bei Verwendung von -1 als Handle-Wert der Rechner "abschmiert". Sicher ist sicher! Wenn Sie das Programm einmal laufen lassen, erscheint zwar unser neues Fenster, genau so, wie wir uns das vorgestellt haben, aber nach dem Betätigen einer Taste passiert nichts. Erst wenn RETURN gedrückt wurde, werden alle zuvor eingegebenen Tasten in dieser Reihenfolge bearbeitet. Wir haben nun ein neues Fenster, leider aber wieder mit dem "Zeilen"-Modus. Ach ja, um das Programm zu verlassen, muß die ESC-Taste betätigt werden und dann, das wissen Sie nun schon, noch einmal die RETURN-Taste.

Aber Gott sei Dank, haben wir in unserer Liste noch den letzten Eintrag "RAW:" zur Verfügung. RAW heißt in Deutsche übersetzt soviel wie "roh", wir erhalten also alle Informationen pur. Aber machen wir uns daraus erst mal nichts, sondern ändern wir "CON:" frohen Muts in "RAW:". Compilieren und linken Sie jetzt diese neue Version nochmal und schauen sie, was nun passiert.

Wieder haben wir ein eigenes Window, das Sie, wie alle anderen Fenster auch, beliebig auf dem Bildschirm herumschieben, ver-

größern und verkleinern können. Darum brauchen wir uns nicht zu kümmern, das erledigt das Betriebssystem. Wählen Sie Ihr neues Window an, können Sie unserem Programm Daten zuführen. Und siehe da, er reagiert auf jeden Tastendruck sofort. Nun aber erscheint die Eingabe in unserem eigenen Window nicht mehr. Das ist die Stelle, an der wir dem Wort "RAW" (roh) Tribut zollen müssen. Diese Aufgabe fällt nun in unseren Zuständigkeitsbereich.

Alle Ausgaben durch printf werden weiterhin auf dem CLI-Fenster ausgegeben. Das ist ja nicht weiter verwunderlich, da dies für unser Programm die Standardausgabe ist. Möchten wir nun etwas in ein eigenes Fenster schreiben, wofür wir ja ein spezielles Handle haben, so muß dementsprechend auch eine Schreibroutine benutzt werden. Anstelle von printf können wir doch fprintf nehmen, das ja die gleichen Funktionen besitzt. Halt! Stop! Alles zurück! Erinnern Sie sich, fprintf ist eine Routine für gepufferte Dateien, und wir haben mit open eine ungepufferte geöffnet. Das funktioniert nur, wenn wir mit fopen das FILE-Handle besorgt hätten. Und da unser Handle lediglich ein int-Wert und kein FILE-Pointer ist, paßt somit unser Schlüssel (handle) nicht zum Schloß (fprintf). Das wäre natürlich traurig, wenn eine so komfortable Funktion wie printf oder fprintf jetzt nicht benutzt werden könnte. In der Bibliothek sitzt aber noch ein Zwillingsbruder der beiden, namens sprintf. Anstatt die aufbereiteten Daten in die Standardausgabe oder in den Puffer zu schreiben, wird alles in einem String abgespeichert. Ein Aufruf dieser Routine sieht dann so aus:

```
char string[200]; /* nicht zu klein */
int test = 4711;

sprintf(string, "Das Ergebnis ist %5d\n", test);
```

Wenn wir jetzt alles in unser privates Window schreiben können, brauchen wir die printf-Funktion nicht wieder im CLI-Fenster herummalen zu lassen. Jetzt haben wir alles zusammen, um das neue Window nach Herzenslust für Ein- und Ausgaben zu benutzen. Hier das Listing:

```
#define ESC 27

main()
{
  int dummy = 0, anz, handle;
  char c, zeile[256];

  handle = open("RAW:50/50/200/60/Mein Programm", 0, dummy);
  printf("Eröffnet %d handle\n", handle);
  if(handle != -1)
    {
      do
        {
          anz = read(handle, &c, 1);
          /* write(handle, &c, 1); nur das Zeichen ausgeben */
          sprintf(zeile, "Zeichen >%c< Code %d\n", c,c);
          write(handle, zeile, strlen(zeile));
        } while(c != ESC);
      close(handle);
    }
}
```

23.8 Umleitungen

So, das klappt doch hervorragend. Allerdings hat das mit Umleitung der Ein-/Ausgabe nichts zu tun, denn wir öffnen ja einfach einen neuen weiteren Kanal für den Datenfluß. Es ist aber gut zu wissen, daß die ganze Sache mit der Umleitung von Daten bereits vom Betriebssystem geregelt wird. Ein Programmierer braucht sich kaum darum zu kümmern. Einzige Voraussetzung dafür, daß die Daten auch umgeleitet werden können, ist, daß man eben die Standardein-/ausgabe benutzt. Darunter fallen ja so bekannte Funktionen wie printf, scanf, putchar, getchar, puts und wie sie alle heißen.

Nehmen Sie sich doch bitte noch einmal die erste "RAW-Version" vor, also die, in der Text noch mittels printf-Funktion ins CLI-Fenster geschrieben wurde. An diesem Beispiel wollen wir uns mit der Umleitung vertraut machen.

Der "normale" Aufruf für Programme, die keine Argumente erwarten, lautet:

```
programmname
```

Nun können wir hinter den Namen der Datei noch Kommandos setzen, die nicht vom Programm, sondern bereits vom Betriebssystem verarbeitet werden. Diese Befehle, durch ">" oder "<" begonnen, teilen dem Betriebssystem mit, daß die Standardein-/ausgabe verändert werden soll. Sehen wir uns hierzu ein praktisches Beispiel an:

```
prg <datei1 >datei2
```

Diese beiden Parameter bekommt das Programm selbst nie zu Gesicht. Sie haben aber große Auswirkungen, da das Betriebssystem die Standardeingabe von der Datei "datei1" erwartet und die Ausgabe in "datei2" schickt. Um Öffnen und Schließen der Dateien braucht man sich nicht zu kümmern, das wird alles vollautomatisch erledigt. Die Größer- und Kleinerzeichen geben die Richtung der Daten an, so etwa, wie ein Pfeil dies signalisiert. Wir können dadurch sofort sagen, daß die Informationen in das File mit dem Namen "datei2" geschrieben werden. Schauen Sie sich diesen Vorgang doch mit unserem RAW-Programm (Version 1) an. Durch die Ausgabe mittels printf sind wir in der Lage, alle Texte in eine Datei oder auch an den Drucker ("PRT:") weiterzuleiten. Starten wir unser Programm (Es heißt hier RAW1!) durch folgende Zeile:

```
raw1 >ausgabe.dat
```

Es erscheint wieder das von uns bestimmte Fenster, allerdings scheint sich nach einem Tastendruck nichts zu bewegen. Das ist auch richtig, denn die Ausgaben, die sonst im CLI-Fenster erschienen, wandern nun direkt in die Datei "ausgabe.dat". Drücken Sie, nachdem Sie ein wenig blind auf der Tastatur herumgetippt haben, die ESC-Taste. Das Fenster verschwindet, Sie befinden sich wieder im CLI und können sich nun die Datei mit dem DOS-Befehl TYPE oder einem Editor (z.B. ED) anschauen. Es stehen genau die erwarteten Ausgaben darin.

Ebensogut könnte man die Eingabe nicht über Tastatur, sondern über eine Datei steuern. Man hätte sich so eine eigene MAKE-Datei geschaffen. Bei unserem Programmbeispiel macht das natürlich keinen Sinn, da wir gar nicht von der Standardeingabe lesen. Dementsprechend nützt auch eine Umlenkung derselben wenig.

Die Standardein-/ausgabe sind ja ganz normale Transferleitungen. Genau wie beim Öffnen einer Datei erhält man ein Datei-Handle. Da diese Leitungen ständig geöffnet sind, brauchen wir uns darum nicht zu kümmern. Nichtsdestotrotz existieren natürlich Variablen für die Handles. Sie heißen:

stdin
stdout
stderr

Das erste ist "standard input" (Standardeingabe) und das zweite "standard output" (Standardausgabe), aber das dritte?

Zusätzlich zur Ein- und Ausgabe von "normalen" Informationen bietet uns C noch einen separaten Fehlerkanal an. Das hat auch seinen Sinn, denn stellen Sie sich doch einmal die folgende Situation vor:

Wie oben geschildert, leiten Sie die Informationen nach Strich und Faden um. Da Sie aber beim Eintippen irgendwo einen Fehler gemacht haben, z.B. nicht existierende Eingabedatei, erhalten Sie eine Fehlermeldung auf dem Bildschirm. Aber Moment, die Standardausgabe wurde doch in eine andere Datei umgeleitet. Das wiederum würde bedeuten, daß auch die Fehlermeldung in dieses File geschrieben wird und Sie von all diesen Geschehnissen nichts mitbekommen würden. Deshalb hat man für Fehlermeldungen noch einen separaten Kanal frei, um trotz diverser Umleitungen dem Benutzer noch Mitteilungen zukommen zu lassen.

Sie können also alle drei Variablen (stdin, stdout und stderr) wie FILE-Pointer benutzen, die Sie über fopen erhalten. Aber Vor-

sicht! Diese gepufferten Ein-/Ausgaberoutinen dürfen Sie nicht mit den ungepufferten verwechseln. Erlaubte Funktionen mit den obigen FILE-Pointern wären fprintf, fwrite, fread usw.

Wenn Sie noch einmal die arg strapazierte Datei "stdio.h" heraussuchen, können Sie dort auch die Definition von getchar als

```
getc(stdin)
```

entdecken. Ich hoffe, daß ich Ihnen mit diesen Beispielen genug Anregungen für weitere Entwicklungen gegeben habe. Weiterhin viel Spaß beim Programmieren in C.

24. Tips und Tricks

Beim Programmieren stößt man bestimmt des öfteren auf Unstimmigkeiten mit dem, was man erwartet hat. Dann steht vor einem viel Arbeit, um herauszufinden, an wem es denn nun liegt, am Programm, am Compiler oder am Betriebssystem. Einige Tips und Tricks rund um das Thema "C" und die Programmierung des AMIGA wurden daher hier aufgeführt.

24.1 Starten von der Workbench

Haben Sie schon einmal versucht, Ihre eigenen C-Programme von der Workbench aus zu starten? Bestimmt werden Sie sich dann gewundert haben, daß trotz voller Diskette nichts im Window dieser Disk zu sehen war. Dies liegt daran, daß jedes Programm noch zusätzlich ein File besitzt, in dem die Daten über Aussehen und Ort des Symbols der Datei untergebracht sind. Daher hat jedes Programm, das irgendwo in einem Fenster erscheint, eine solche Datei mit der Endung ".info". Auch für unser Programm brauchen wir so eine Datei um es sichtbar zu machen. Genau zu diesem Zweck suchen wir uns nun auf der Workbench eine geeignetes Icon heraus. Vielleicht nehmen Sie die Uhr oder "Notepad". Natürlich können Sie auch jedes andere Symbol als Vorlage für Ihr Programm nehmen. Wir kopieren es also mit

```
copy notepad.info test-workb.info
```

(Unser kommendes Programm wird "test-workb" lauten.)

Wenn wir uns nun das Inhaltsverzeichnis mit unserem Icon vornehmen (vielleicht ist es ja die RAM-Disk?), so setzen wir es erst einmal an eine freie Stelle im Fenster. Im Moment hat es ja noch die gleiche Position wie das Original. Anschließend speichern wir diesen Zustand mit "Special-Snapshot" auf der Workbench ab. Jetzt brauchen wir nur noch ein geeignetes Programm hierfür. Theoretisch könnte man fast jedes zuvor geschriebene verwenden, praktisch soll aber ein speziell hierfür ausgelegtes benutzt werden. Schauen Sie sich es zuerst einmal an:

```
#include "stdio.h"


main(argc, argv)
int argc;
char *argv[];
{
  int dummy = 0, handle;
  char zeile[256];

  if(argc)   /* Argumentzahl ungleich 0 */
    {
      handle = open("RAW:50/50/200/60/Mein Programm", 0, dummy);
      if(handle != -1)    /* Kein Fehler beim Öffnen */
        {
          sprintf(zeile, "Vom CLI gestartet!\n");
          write(handle, zeile, strlen(zeile));
          while(--argc >= 0)
            {
              write(handle, *argv, strlen(*argv));
              argv++;
            }
          read(handle, zeile, 1);    /* Tastendruck abwarten */
          close(handle);
        }
      else
        fprintf(stderr, "\nFehler beim Eröffnen des Fensters\n");
    }
  else
    {
      printf("Von der Workbench gestartet!\nRETURN-Taste!\n");
      getchar();
    }
}
```

In diesem Programm sind beim Umgang mit der Workbench einige Besonderheiten zu beachten. Zuerst ist es für den Programmierer doch wichtig zu wissen, ob das Programm von der Workbench oder vom CLI gestartet wird (Warum, das wird gleich erklärt!). Dies kann man an argc, also dem Argumentzähler im Neudeutschen, ablesen. Ist er gleich 0, so wurde das Pro-

gramm von der Workbench aus gestartet. Dieser Konvention wurde zumindest beim Lattice nachgegangen! Und das ist auch sehr gut so. Überlegen wir uns, welche Werte beim Aufruf vom CLI möglich sind! Minimal enthält argc 1, nämlich genau dann, wenn nur der Programmname ohne jegliche Parameter beim Aufruf verwendet wird. Andernfalls erhöht sich diese Variable um die Anzahl der Parameter. Die 0 wird nie benutzt, also eine ideale Möglichkeit, diese beiden verschiedenen Aufrufe zu unterscheiden. Stellt sich die Frage: "Wen interessiert es denn, von wo sein Programm gestartet wird. Das ist doch völlig egal!"

Genau das ist es eben nicht! Starten Sie beispielsweise unser letztes Programm (RAW2), so wird von uns ein Fenster geöffnet. Gleichzeitig erhalten wir von der Workbench
ebenfalls ein Window zur Verfügung gestellt, mit dem wir nun gar nichts anfangen können. Ein Fenster wird benutzt, das andere steht leer im Raum. Deshalb muß man unterscheiden zwischen einem Start

- von der Workbench
- vom CLI

Im ersten Fall erhalten wir automatisch ein Fenster - und jetzt kommt das Beste - bei dem die Stardardein-/ und -ausgabe bereits auf dieses Window gelenkt wird. Sie können also mit allen Grundfunktionen wie scanf und printf arbeiten und kommen trotzdem in den Genuß eines eigenen Windows.

Im zweiten Fall, das haben wir bereits abgehandelt, müssen wir uns um unser Fenster und die dazugehörige Datenein- und -ausgabe selbst kümmern.

24.2 Weitere Anweisungen des Präprozessors

Die Anweisungen "#define" und "#include" kennen wir und haben sie schon reichlich benutzt. Es gibt aber noch eine Reihe anderer Präprozessor-Befehle, die dem Programmierer die Arbeit erleichtern können. Die wichtigsten sind folgende:

```
#undef MAKRO
```

Die ist genau das Gegenteil von "#define". Die Definition des Makros wird aufgehoben und ist ab dieser Stelle in der Datei wieder unbekannt. Nehmen wir aber an, Sie hätten zwei "#define"-Anweisungen für einen Ersatz benutzt, beispielsweise

```
#define EOS 0
...
#define EOS '\0'
```

Es ist klar, daß in allen nachfolgenden Programmteilen die letzte Definition von EOS verwendet wird. Die erste Zuweisung ist aber nicht vergessen, sie ist momentan nur unsichtbar. Man kann diesen Vorgang gut mit lokalen Variablen vergleichen, die sich durch den gleichen Namen überdecken. Man kann immer nur auf die zuletzt definierte Variable (oder hier Define-Definition) zugreifen. Wenn man nun allerdings mit "#undef" die Definition widerruft, z.B.

```
#undef EOS
```

so ist das define EOS immer noch vorhanden, jetzt aber mit seiner ersten Zuweisung, nämlich "0".

```
#ifdef MAKRO
```

Durch diese Anweisung kann man, abhängig davon, ob das Makro bereits definiert wurde, bestimmte Teile der Datei compilieren. Sollte das Makro definiert gewesen sein, so wird der folgende Teil bis zu den Präprozessor-Befehlen

```
#elseif MAKRO
```

oder

```
#endif MAKRO
```

vom Compiler bearbeitet. Ist das Makro an dieser Stelle jedoch unbekannt, so werden alle Zeilen bis zu den genannten folgenden Befehlen übersprungen. Sollte hier nun "#elseif" auftauchen, so wird jetzt der folgende Quelltext bis "#endif" compiliert.

Diese Anweisungen sind also den C-Befehlen

```
if()
  {
    ...
  }
else
  {
    ...
  }
```

vergleichbar, nur, daß diese Befehle nichts mit dem endgültigen Code zu tun haben. Die Umkehrung dieses Vorganges ermöglicht die Zeile

```
#ifndef MAKRO
```

Hier darf das Makro noch nicht definiert sein, damit der folgende Teil compiliert wird. Man kann somit bestimmte Teile einer Datei durch Setzen eines Defines einbinden oder auch herauslassen, ohne die Datei im größeren Umfang zu verändern. Wo werden diese Anweisungen sinnvoll eingesetzt?

Bei Programmen, die auf verschiedenen Systemen (unterschiedliche Compiler oder gar Rechner) compiliert werden sollen, müssen ab und zu Besonderheiten, wie z.B. Compilerfehler berücksichtigt werden. Um zu einem fehlerlosen Compilerlauf zu kommen, muß dann nur noch ein Define gesetzt werden, das z.B. den Compilertyp angibt. In einigen Header-Dateien, die den Compilern mitgeliefert werden, können Sie diese Befehle wiederfinden.

24.3 Fehlerursachen und deren Beseitigung

Kein Programm wird von Anfang an ohne Fehler (syntaktisch oder logisch) sein. Die syntaktischen Unstimmigkeiten werden uns in der Regel vom Compiler mitgeteilt, so daß deren Beseitigung keinerlei Probleme machen sollte. Dennoch kann es passieren, daß man mit der Meldung scheinbar nichts anfangen kann. Deshalb werden hier mögliche Fehlerquellen näher erläutert, um Sie vor allzu großen Kopfschmerzen zu bewahren.

- *fehlendes Semikolon*

  Sollte das Semikolon nicht gerade in der angegebenen Zeile fehlen, so findet man die Stelle meist ein bis zwei Zeilen darüber.

- *geschweifte Klammern stimmen nicht überein*

  Dieser Fehler erfordert oft, den gesamten Bereich zuvor (in der noch kein Fehler aufgetaucht ist) nach diesen Klammern zu kontrollieren. Fehlt nämlich irgendwo eine Klammer, so wird der folgende Teil zu der bislang beschriebenen Funktion hinzugerechnet. Dies führt wiederum zu sehr mysteriösen Fehlermeldungen, bspw. das fehlende Semikolon bei der folgenden Funktionsdefinition.

- *falsche Ergebnisse*

  Wenn eine einwandfrei laufende Funktion plötzlich falsche Resultate liefert, so kann der Fehler darin liegen, daß man vergessen hat, die Funktion zu deklarieren (nicht bei int). Deshalb sollte man alle Funktionen, die keine int-Rückgabewerte benutzen, zu Beginn der Datei global deklarieren. Bei neu hinzugefügten Funktionen, die diese Routine ebenfalls verwenden, kann man dann diesen Fehler nicht mehr machen.

Komplexe Formeln, die mit vielen verschiedenen Operatoren und Datentypen arbeiten, sollte man generell klammern. Dies erhöht nicht nur die Übersichtlichkeit, sondern beugt auch ungewollten Fehlern vor. Kaum jemand kennt alle Prioritäten der diversen Operatoren auswendig, so daß es dort schnell zu Problemen kommen kann, wenn man sich mal irrt. Lieber ein paar Klammern zuviel verwendet, als ein Paar zu wenig.

Passen Sie immer auf die leicht zu verwechselnden Kombinationen

== und =
&& und &
|| und |

auf. Auch bei geübteren Programmierern können sich noch solche "banalen" Fehler einschleichen.

Beachten Sie die zulässigen Maximalwerte der unterschiedlichen Datentypen bei Berechnungen (kann von Compiler zu Compiler verschieden sein!):

char -128 bis 127
int -32767 bis 32766
Long -2147483648 bis 2147483647
float -10^{38} bis 10^{38}
double -10^{303} bis 10^{303}

Diese dürfen nie überschritten werden, auch nicht in Zwischenergebnissen einer längeren Formel wie:

```
i = (x * y * z - z) / y - x;
```

Dies kann bei "x * y * z" schnell vorkommen, wenn die drei Faktoren große Zahlen beinhalten.

- *Programmabsturz*

  Dies kann vielerlei Ursachen haben, z.B.

  - nicht initialisierte Pointer
  - falsche Parameterübergabe (Verwechslung der Datentypen)
  - Pointer-Zugriff auf ungerade Adresse (außer char-Pointer)

25. Systemprogrammierung

Einer der Gründe, warum wir uns bislang mit C beschäftigt haben, ist die Geschwindigkeit dieser Sprache. Ein Hauptgrund aber, besonders für Amiga-Besitzer, ist die Tatsache, daß das Betriebssystem des Amiga in C geschrieben ist und deshalb auch alle Besonderheiten dieser Sprache berücksichtigt werden. Dies macht sich natürlich erst dann richtig bemerkbar, wenn man systemnah programmiert, um die besonderen Leistungen aus dem Rechner herauszuholen.

Wir wollen uns nun ein bißchen in der Welt von Intuition umsehen, einem der faszinierendsten Gebiete der Programmierung. Intuition stellt den Teil des Betriebssystems dar, der sich unter anderem mit Windows, Screens, Icons, Menüs und Maus beschäftigt. Sie geben den Programmen einen professionellen Touch. Allerdings sind die Möglichkeiten von Intuition so umfassend und vielfältig, daß wir nur ein wenig in die Materie hineinriechen können. Sollten Sie dann Geschmack gefunden haben, empfehle ich Ihnen weiterführende Literatur. Wir beschränken uns auf den Einstieg in dieses Gebiet, und zwar auf die Konstruktion von Windows und Screens.

Eine kleine Warnung vorweg: Intuition ist sehr komplex. Deshalb wird es jetzt etwas komplizierter. Auch, wenn Sie nicht alles verstehen, sollten Sie die Programme ausprobieren und ein wenig variieren.

25.1 Das Intuition-Prinzip

Möchte man Intuition in eigenen Programmen verwenden, so muß man bestimmten Gepflogenheiten folgen. So kann man z.B. erst dann Routinen von Intuition benutzen, wenn man die Library "Intuition" geöffnet hat. Intuition ist nichts anderes als eine große Library mit den Funktionen, die im Zusammenhang mit Windows und dergleichen benutzt werden. Der Unterschied zu den bisher bekannten Bibliotheken unseres C-Compilers besteht darin, daß diese Routinen nicht während des Linkens zusammengebunden werden. Man verläßt sich darauf, daß sie ir-

gendwo im Rechner gelagert werden und während des Programmlaufes zur Verfügung stehen. Das hat einige Vorteile. Da viele Programme oft dieselben Intuition-Routinen benötigen, brauchen diese Routinen ja nicht mehrfach im Speicher stehen. Jedes Programm kann, auch wenn es gleichzeitig mit anderen im Speicher arbeitet (Multitasking), diese Funktionen zu eigenen Zwecken benutzen. Das spart Arbeitsspeicher. Dabei ist es völlig egal, wo diese Routinen gerade im Speicher untergebracht sind. Öffnet man die "intuition.library", so erhält man einen Zeiger auf den Beginn dieses Funktionspaketes zurück. Und hier sind wir bereits mitten in der Programmierung. Ein Markenzeichen der Intuition-Programmierung sind eine Menge Zeiger und Strukturen. Strukturen werden wir daher fast bis zum Exzeß benutzen, weshalb ich Sie bitten möchte, vielleicht nochmals im entsprechenden Kapitel nachzulesen.

25.2 Ein Window unter Intuition

Nanu, werden Sie vielleicht denken, ein Window hatten wir doch bereits auf dem Bildschirm. Das stimmt zwar, aber dies wurde mit dem normalen Open-Befehl erledigt. Dieser läßt uns zwar die Größe und Position vorwählen, und auch mit dem Verschieben des Fensters klappt alles bestens, doch die besonderen Features werden wir nur unter Intuition einsetzen können. Die Möglichkeiten sind so vielfältig, daß man auf den ersten Blick fast davon erschlagen wird. Aber wagen wir mal den Sprung ins kalte Wasser und betrachten die nötige Window-Struktur, die alle wichtigen Informationen bezüglich des Fensters aufnimmt.

```
struct NewWindow
{
   SHORT LeftEdge, TopEdge;
   SHORT Width, Height;
   UBYTE DetailPen, BlockPen;
   ULONG IDCMPFlags;
   ULONG Flags;
   struct Gadget *FirstGadget;
   struct Image *CheckMark;
   UBYTE *Title;
   struct Screen *Screen;
   struct BitMap *BitMap;
```

```
    SHORT MinWidth, MinHeight;
    SHORT MaxWidth, MaxHeight;
    USHORT Type;
}
```

Diese Strukturdefinition ist übrigens im Binding "intuition/intuition.h" zu finden. Das ist eine ganze Menge für den Anfang! Filtern wir deshalb die Dinge heraus, die wir auch wirklich einsetzen. Der Name ist schon bemerkenswert: NewWindow. Es existiert zwar im gleichen Binding auch eine Struktur Window, wir benötigen aber zur Definition eines eigenen Fensters die Struktur NewWindow.

Die ersten vier Einträge stellen die linke obere Ecke und deren Ausmaße in der Höhe und in der Breite dar. Es sind die gleichen Werte, die wir auch beim Open-Befehl wie

```
open("CON:20/40/200/50/Windowtitel", 0, 0);
```

eingesetzt haben. Um die gleichen Werte für unser Intuition-Fenster zu bekommen, werden die Werte einzeln den Strukturkomponenten zugewiesen. Dabei setzen wir die folgende Definition voraus:

```
struct NewWindow NewWindow;

...

NewWindow.LeftEdge = 20;
NewWindow.TopEdge  = 40;
NewWindow.Width    = 200;
NewWindow.Heigth   = 50;
```

Jetzt kommen schon Möglichkeiten, die kein Open-Befehl mehr leistet, die Einstellung der Farben für unser Fenster.

Der Titel des Fensters und die Querbalken in gleicher Höhe werden durch den Wert in DetailPen bestimmt. Die Zahl beschreibt den Index des Farbregisters. Das muß man sich so vorstellen: Wenn wir 4 mögliche Farben zur Verfügung haben, so sind diese von 0 bis 3 durchnumeriert. Die Farben selbst können ohne weiteres nicht geändert werden; man benutzt also die über

das "Preferences"-Programm voreingestellten Farbzusammenstellungen. Die Hintergrundfarbe belegt dabei stets das Register 0. Ähnliches gilt auch für die zweite Registerangabe: BlockPen. Mit dieser Farbe werden alle Umrandungen des Fensters gezeichnet.

25.2.1 Die Window-Flags

Den folgenden Eintrag (IDCMPFlags) wollen wir überspringen, da er noch nicht gebraucht wird. Dann aber, im Element Flags der NewWindow-Struktur geht es gleich richtig zur Sache. Hier legt man jetzt durch Flags, die mittels Defines gesetzt werden, bestimmte Dinge des Fensters fest. Jedes Defines wie z.B. WINDOWSIZING bestimmt, ob eine Intuition-Funktion benötigt oder nicht benötigt wird. Am besten sehen wir uns die im anschließend aufgeführten Beispielprogramm benutzten Defines in einer kleinen Aufstellung an. Das sind aber längst noch nicht alle. Für unsere Belange sind die hier eingesetzten Defines wohl aber das höchste aller Gefühle.

- *Smart Refresh*

  Hierdurch wird der Amiga veranlaßt, alles, was die Veränderung des Fensters und dessen Inhalt von außen betrifft, zu kontrollieren und zu sichern. Schieben wir beispielsweise ein anderes Window auf unser eigenes, so wird ja vorübergehend ein Teil des Fensters verdeckt. Der Rechner speichert diesen Bereich automatisch in einen Puffer und stellt den Ausschnitt bei Bedarf wieder her. Um das Fenster brauchen wir uns also keine Sorgen mehr zu machen.

- *Activate*

  Beim Öffnen des Fensters wird es sogleich aktiviert. Das erspart uns das erste Anklicken des Windows.

- *Windowsizing*

  erlaubt, das Fenster mit dem Größen-Gadget in seinen Ausmaßen zu ändern. Durch Anwahl dieses Defines erscheint auch das dazugehörige Gadget in der rechten unteren Ecke.

Die Überwachung dieses System-Gadgets wird ebenfalls vom Betriebssystem übernommen.

- *Windowrag*

  Das Fenster kann verschoben werden.

- *Windowdepth*

  Das Window kann mit den zwei Gadgets in der rechten oberen Ecke vor oder hinter andere Windows gebracht werden. Auch hier wird alle damit verbundene Arbeit ohne unser Zutun vom Rechner übernommen.

- *Nocarerefresh*

  Man kann praktisch für alle Dinge, die beim Programmablauf auftreten können, vom Betriebssystem eine Nachricht erhalten. Möchte man unterrichtet werden, wenn z.B. der Benutzer die Größe verändert und damit ein Neuzeichnen des Windows erforderlich wird, so kann man auch das bekommen. Denken Sie beispielsweise an Ihren Editor ED. Ist ein Text in Bearbeitung, der gerade im Window erscheint, so wird nach jeder Veränderung der Fenstergröße das Window neu beschrieben. Das Programm erhält also die Mitteilung, daß das Fenster auf den aktuellen Stand gebracht werden soll. Mit obigen Define teilen Sie aber dem Betriebssystem mit, daß Sie keine solchen Meldungen wünschen. Wir brauchen die Mitteilungen auch nicht, weil wir gleichzeitig SMART_REFRESH eingeschaltet haben. Und dadurch werden wir von solchen Update-Arbeiten entlastet.

Von den folgenden Elementen der NewWindow-Struktur interessieren nur noch "Title" sowie die letzten fünf. Wie der Name schon sagt, wird in Title der Titel des Windows eingetragen, der in der oberen Leiste erscheint. Wenn Sie sich das folgende Programm ansehen, entdecken Sie, daß bei der Zuweisung:

```
NewWindow.Title = "Das eigene Window";
```

lediglich die Adresse des Strings zugewiesen wird. Sollten Sie keine Konstanten (wie im Beispiel oben) verwenden, so müssen Sie selbst den Speicherplatz dafür bereitstellen und nur dessen Beginn an den Eintrag "Title" zuweisen.

Jetzt wird es wieder einfacher. Die Werte MinWidth, MinHeight, MaxWidth, MaxHeight geben feste Minimal- und Maximalwerte für unser Fenster an. Der Anwender wird darin gehindert, diese Grenzen zu über- beziehungsweise zu unterschreiten. Die Maximalhöhe eines Fensters beträgt übrigens 256 oder 512 (je nach Modi) Punkte, da wir in der Regel einen PAL-Amiga vor uns haben. Die amerikanischen Versionen schaffen dagegen "nur" 200 oder 400 Punkte, weswegen diese Zahlen des öfteren an dieser Stelle auftauchen.

Zuletzt tragen wir noch WBENCHSCREEN in das Element "Type" ein, damit wir die von der Workbench voreingestellten Parameter benutzen können.

25.2.2 Öffnen eines Windows

Nach diesen ganzen Vorarbeiten kann nun endlich das Window geöffnet werden. Dies geschieht mit der Funktion OpenWindow, die uns auch einen Zeiger zurückgibt. Dieser zeigt auf eine Window-Struktur, die nicht mit NewWindow verwechselt werden sollte. Sie ist noch einiges umfangreicher als "NewWindow" und kann in "intuition.h" begutachtet werden.

Die OpenWindow-Funktion benötigt die Adresse unserer NewWindow-Struktur als Parameter. Da es bei vielen Compilern bereits möglich ist, ganze Strukturen zu übergeben, verwenden wir unbedingt den Ausdruck

```
&NewWindow
```

zur Bestimmung der Adresse.

Ist dann alles ordnungsgemäß abgelaufen, hat man ein eigenes Window genau nach den in "NewWindow" eingetragenen Angaben auf dem Bildschirm.

Um das Window wieder zu schließen, z.B. wenn das Programm beendet wird, genügt der Aufruf der Funktion CloseWindow, die den Window-Zeiger als einzigen Parameter erhält. Schon ist auch das Fenster wieder verschwunden.

Zuletzt schließen wir auch die Intuition-Library wieder, damit wir alles so zurücklassen, wie wir es vorgefunden haben. Denken Sie immer daran, die Dinge zuerst zu schließen, die Sie zuletzt geöffnet haben. Schließen Sie z.B. Intuition vor dem letzten Window, so beendet der Guru Ihr Programm schneller als erwartet.

25.2.3 Ein Window-Programm

Die ganze graue Theorie finden Sie in dem schon angekündigten Listing wieder. Das Programm öffnet ein Fenster, das Sie für eine bestimmte Zeit verschieben, vergrößern und verkleinern, vor und hinter andere Fenster bewegen können.

```
#include <exec/types.h>
#include <intuition/intuition.h>

extern struct Window *OpenWindow(); /* Saubere Deklaration */
extern long *OpenLibrary();         /* Hallo Aztek-User! */
struct IntuitionBase *IntuitionBase;


#define INTUITION_REV 0


main()
{
   struct NewWindow NewWindow;
   struct Window *Window;
   long i;

   IntuitionBase = (struct IntuitionBase *)
      OpenLibrary("intuition.library", INTUITION_REV);
```

```
    if(IntuitionBase == NULL)
       exit(FALSE);

    NewWindow.LeftEdge  = 20;
    NewWindow.TopEdge   = 20;
    NewWindow.Width     = 200;
    NewWindow.Height    = 80;
    NewWindow.DetailPen = 0;
    NewWindow.BlockPen  = 2;
    NewWindow.IDCMPFlags = NULL;
    NewWindow.Flags = SMART_REFRESH | ACTIVATE
       | WINDOWSIZING | WINDOWDRAG | WINDOWDEPTH | NOCAREREFRESH;
    NewWindow.FirstGadget = NULL;
    NewWindow.CheckMark   = NULL;
    NewWindow.Title       = (UBYTE *)"Das eigene Window";
    NewWindow.Screen      = NULL;
    NewWindow.BitMap      = NULL;
    NewWindow.MinWidth  = 80;
    NewWindow.MinHeight = 25;
    NewWindow.MaxWidth  = 640;
    NewWindow.MaxHeight = 256;
    NewWindow.Type = WBENCHSCREEN;

    if((Window = OpenWindow(&NewWindow)) == NULL)
       exit(FALSE);
    for(i = 0; i < 800000; i++)  /* Kleine Pause */
       ;

    CloseWindow(Window);
    CloseLibrary(IntuitionBase);
    exit(TRUE);
}
```

Das obige Programm folgt genau den zuvor gemachten Angaben und Anforderungen. Zuerst wird die Intuition-Library geöffnet. Falls dies aus irgendeinem Grunde nicht möglich war, erhalten wir als Intuition-Zeiger 0 zurück. Ein Weiterarbeiten hat dann keinen Zweck mehr, und wir beenden an dieser Stelle das Programm. Auch der zurückgelieferte Window-Pointer kann Null sein, weswegen er überprüft werden sollte.

Beachten Sie bitte die Deklaration der Funktion OpenWindow am Anfang des Listings. Dadurch wird nicht nur ein guter C-Stil beibehalten, sondern es wird auch das "Zurechtbiegen" von

Rückgabewerten mittels der cast-Anweisung unterbunden. Dies ist z.B. bei OpenLibrary der Fall, da diese Routine nicht nur Intuition-Pointer, sondern noch diverse andere Zeiger-Typen liefern kann. Trotzdem sollte diese Funktion zumindest als Routine deklariert werden, die Zeiger - welcher Art auch immer - zurückgibt. Dies verhindert z.B. beim Aztek eigentümliche Systemabstürze, die daraus resultieren, daß die Funktion nicht deklariert wurde. Das bedeutet für den Compiler, es werden Integer-Werte geliefert, die beim Aztek nur zwei Byte lang sind. Daß der übermittelte Vier-Byte-Wert nicht ankommt, ist klar. Es werden nur zwei Byte entgegengenommen.

Durch die cast-Anweisung "(struct IntuitionBase *)" wird dann auch der gründlichste Compiler überlistet. Meistens ist der dann entstandene Wert auch noch ungleich 0 und veranlaßt das Programm nicht zum Abbruch. Bei dem nächsten Zugriff auf eine Library-Funktion über dessen Zeiger IntuitionBase fällt das System dann böse auf die Nase. Dem Lattice kann dieses Unglück übrigens nicht passieren, da seine Integer stets vier Bytes in Anspruch nehmen. Aber auch hier sollte die Devise (bei vertretbarem Aufwand) sein: Sicher ist sicher!

Beim Öffnen der Library möchte die Routine noch eine Versionsnummer, die für die korrekte Abarbeitung mindestens erforderlich ist. Wenn das System die gleiche oder eine spätere Version besitzt, geht alles klar. Da unser Programm keine besonderen Wünsche hat, wählen wir 0.

Beim Öffnen des Windows wird die Adresse der NewWindow-Struktur übergeben. Das System übernimmt dann die Daten in einen internen Bereich, so daß nach dieser Operation die Variable NewWindow nicht mehr benötigt wird. Wenn wir etwas mit dem Fenster anstellen wollen, wird das über die Window-Struktur gemanagt.

Falls Sie beim Compilieren eine Reihe von "Warnings" erhalten haben, so ist dies nicht weiter schlimm. Einige Strukturen, die von uns zwar nicht explizit benutzt werden, aber trotzdem in einer Struktur-Definition als Unterelement auftauchen, sind dann noch nicht definiert. Sie können sich ja mal den Spaß machen,

die dazu nötigen Bindings durch "include" miteinzubinden. Dabei habe ich allerdings die Erfahrung gemacht, daß durch die neue Definition weitere unbekannte Strukturen auftauchen, die wiederum definiert sein wollen. So bleibt einem schließlich nichts anderes übrig, als nahezu alle Bindings zu "includen". Daraus resultiert aber ein enorm hoher Speicherplatzaufwand, und die Compilierzeit wird erheblich verlängert. Ich würde deshalb davon abraten, es sei denn, Sie besitzen eine gigantische RAMDisk und haben dort alle Include-Files deponiert. Am Objectcode ändert sich dadurch jedenfalls nichts.

Es ist ratsam, nachdem das Rohgerüst eines Window-Programms einmal steht, hier nun ein bißchen zu experimentieren. Ändern Sie doch mal einige Werte in der NewWindow-Struktur nach eigenem Gusto, um deren Auswirkungen auf das Fenster zu betrachten. Dabei sollten Sie aber die noch unbekannten Struktureinträge sowie das Type-Element unberührt lassen.

Es ist hier nicht möglich, den Bereich "Windows unter Intuition" auch nur annähernd erschöpfend darzustellen. Wie erwähnt, sollten Sie sich spezielle Fachliteratur zu diesem Gebiet zulegen, wenn Sie mehr darüber wissen möchten. Wenden wir uns nun einem weiteren Intuition-Bereich zu: den Screens.

25.3 Screens

Ein Screen ist nichts anderes als ein (Bild-)Schirm. Bei den herkömmlichen PCs und Homecomputern gibt es stets nur einen Screen, der das enthält, was auf dem Monitor zu sehen ist. Beim Atari ST hat man z.B. einen Screen (auch wenn man das dort nicht so nennt), auf dem mehrere Fenster dargestellt werden können. Genauso verhält es sich beim Amiga. Einen Screen kennen Sie schon gut, den Workbench-Screen. Auf jedem Screen können fast beliebig viele Fenster geöffnet werden (hängt vom Speicherplatz ab). Er bestimmt auch die Farbzusammenstellung und die Anzahl der zur Verfügung stehenden Farben, die von den Fenstern eingesetzt werden können. Die Workbench bietet in

der Regel 4 verschiedene Farben und arbeitet mit einer Auflösung von 640 * 256 Punkten. Dies sind auch die Vorgaben für alle darauf erstellten Fenster.

Genau diese Werte kann man aber auch per Programm selbst bestimmen und sich einen eigenen Screen "zurechtstricken". Die Farbenanzahl hängt von der Zahl der verfügbaren Bitmaps ab. Eine Bitmap stellt einen Teil des Speichers dar, der zu Speicherung der Grafik benutzt wird. Je mehr Bitmaps man einrichtet, desto mehr Farben sind möglich, desto mehr Speicherplatz wird aber auch belegt. Die Workbench hat bei 4 Farben 2 Bitmaps. Eine Tabelle macht deutlich, wie Farbenanzahl und Bitplanes zusammenhängen.

```
Zahl der Bitplanes -> Zahl der Farben
1 -> 2
2 -> 4
3 -> 8
4 -> 16
5 -> 32 (nicht immer anwendbar)
```

Man kann aber nicht nur die Anzahl der Farben wählen, sondern auch die Auflösung des Screens. Bis zu 32 Farben (ohne besondere Tricks) und eine Auflösung von 640 * 512 Punkten sind so maximal möglich. Wie auch beim Window können Sie zwei Farbregister belegen, die für das Zeichnen der Ränder und deren Hintergrund zuständig sind. Da uns diese Werte, die in der NewScreen-Struktur einzutragen sind, ziemlich stark an die NewWindow-Struktur erinnern, hier erstmal die Strukturdefinition:

```
struct NewScreen
{
   SHORT LeftEdge, TopEdge, Width, Height, Depth;
   UBYTE DetailPen, BlockPen;
   USHORT ViewModes;
   USHORT Type;
   struct TextAttr *Font;
   UBYTE *DefaultTitle;
   struct Gadgets *Gadgets;
   struct BitMap *CustomBitMap;
}
```

Die ersten Angaben in dieser Struktur heißen nicht nur genauso wie die in "NewWindow", sie haben auch exakt die gleiche Bedeutung. "Depth" gibt die schon besprochene Anzahl der Bitplanes an (1-5), und "DetailPen" und "BlockPen" stellen die bekannten Farbindices dar. Sie richten sich nach der Anzahl der zur Verfügung stehenden Bitplanes.

Die nächsten wichtigen Einträge sind Type (Hier muß das COSTOMSCREEN-Define gesetzt werden) und DefaultTitle, der auf die Titelzeile des Screens zeigt. Das reicht bereits für unser Programm aus, um einen Screen vollständig zu definieren.

Das schöne an Intuition ist, daß alles einem bestimmten Strickmuster folgt, so daß sich viele verschiedene Probleme auf ein und dieselbe Weise lösen lassen. Wenn Sie die Vorgehensweise zum Erstellen eines Windows verstanden haben, dürfte es auch kein Problem sein, einen Screen auf den Bildschirm zu zaubern. Zuerst wird die NewScreen-Struktur in der oben angegebenen Weise belegt. Darauf wird der Screen mit der Funktion:

```
Screen = OpenScreen(&NewScreen);
```

geöffnet. Die Variable "Screen" stellt einen Zeiger auf eine Struktur namens "Screen" dar. Auch hier muß man die Strukturen NewScreen und Screen schön auseinanderhalten. NewScreen wird nur ein einziges Mal für die OpenScreen-Funktion benötigt. Die Daten werden in die Screen-Struktur übertragen (Die Screen-Struktur ist wesentlich umfangreicher als NewScreen!), und als Rückgabewert erhält man den Zeiger auf die neue Screen-Struktur.

Um ein Fenster auf diesem Screen zu öffnen, muß die Initialisierung der NewWindow-Struktur ein wenig verändert werden. Zum einen wird in "Type" das Define WBENCHSCREEN durch CUSTOMSCREEN ersetzt, zum anderen muß der Eintrag "Screen" mit dem Screen-Pointer versorgt werden. Das Define CUSTOMSCREEN sagt aus, daß wir das Fenster auf unserem eigenen Screen öffnen wollen. Dadurch erhält das Window alle Möglichkeiten, die unser Screen anbietet. Da aber auch mehrere

Screens von einem Programm geöffnet werden können, muß das Window an einen bestimmten "angehängt" werden. Diese Zuweisung kann man daher erst dann machen, wenn der Screen bereits geöffnet und man so im Besitz des Screen-Pointers ist.

Vor dem Programmende wird selbstverständlich der Screen wieder mit CloseScreen geschlossen, dem der Screen-Zeiger übergeben wird. Denken Sie daran, daß das Window vor dem Screen geschlossen werden muß. Was sonst passiert, brauche ich Ihnen ja wohl nicht zu sagen.

25.3.1 Ein Screen-Programm

Das Listing zum Thema Screens hat noch ein paar Erweiterungen, die gleich erläutert werden.

```
#include <exec/types.h>
#include <intuition/intuition.h>

extern LONG OpenLibrary();
extern struct Screen *OpenScreen();
extern struct Window *OpenWindow();


struct IntuitionBase *IntuitionBase;

#define INTUITION_REV 0


struct NewScreen NewScreen =
{
   0,0,
   640,  /* Breite */
   256,  /* Dt. Version 256 Zeilen */
   3,    /* 3 Bitplanes = 8 Farben */
   3,5,  /* mal ¿ne andere Farbkombination */
   HIRES,
   CUSTOMSCREEN,
   NULL,
   "Zum Programmende bitte Close-Gadget anklicken!",
   NULL,
   NULL,
};
```

```
struct NewWindow NewWindow =
{
   40, 40,   /* X- und Y-Position */
   280, 120, /* Breite, H he */
   4, 6,     /* Farben (0-7) */
   CLOSEWINDOW,
   WINDOWCLOSE | SMART_REFRESH | ACTIVATE
      | WINDOWSIZING | SIZEBRIGHT | WINDOWDRAG | WINDOWDEPTH,
   NULL,
   NULL,
   "*** Hallo ***",
   NULL,    /* Hier kommt sp ter der Screen-Pointer rein */
   NULL,
   190, 20,
   640, 256,
   CUSTOMSCREEN
};


main()
{
   struct Screen *Screen;
   struct Window *Window;

   if((IntuitionBase = (struct IntuitionBase *)
      OpenLibrary("intuition.library", INTUITION_REV)) == NULL)
      exit(FALSE);

   if( (Screen = OpenScreen(&NewScreen) ) == NULL)
      exit(FALSE);

   NewWindow.Screen = Screen;  /* Nicht vergessen! */

   if( (Window = OpenWindow(&NewWindow) ) == NULL)
      exit(FALSE);

   /* Auf Close-Gadget warten */
   Wait(1 << Window->UserPort->mp_SigBit);

   printf("\nLetzte Windowwerte: %d/%d/%d/%d\n\n",
      Window->LeftEdge,
      Window->TopEdge ,
      Window->Width   ,
      Window->Height  );
```

```
    CloseWindow(Window);     /* Alles wieder der Reihe nach
schliemen */
    CloseScreen(Screen);
    CloseLibrary(IntuitionBase);
    exit(TRUE);
}
```

Da C-Programmierer ziemlich tippfaul sind, werden Strukturinitialisierungen am besten gleich bei der Definition der Variablen durchgeführt. Eine weitere Neuerung stellt der Eintrag in "IDCMPFlags" dar: CLOSEWINDOW. In der Kombination mit dem ebenfalls neu hinzugekommenen WINDOWCLOSE in "Flags" kann man das Close-Gadget abfragen. Mit dem etwas unübersichtlich ausschauenden Befehl:

```
Wait(1 << Window->UserPort->mp_SigBit);
```

wartet dann das System auf die in "IDCMPFlags" eingetragenen Ereignisse. In unserem Fall ist die einzig mögliche Situation, daß der Benutzer das Close-Gadget angeklickt hat. Wie bei unserem Window-Programm auch sind ja alle Flags gesetzt, die dem User die Veränderung der Größe und der Position des Windows erlauben. In einigen Fällen können diese Werte für das Programm interessant sein. Über den Window-Pointer zeigen wir auf die gewünschten Elemente LeftEdge, TopEdge, Width und Height. So einfach ist das!

Weil das obige Beispielprogramm 3 Bitplanes benutzt, sind maximal 8 Farben zugänglich. Die Farbregister können deshalb von 0 bis 7 reichen. Wenn Sie nun selbst ein wenig mit Screens herumspielen, sollten Sie bedenken, daß der Speicherplatz sehr schnell knapp werden kann. Allein das kleine oben abgedruckte Programm benötigt knapp 80 K Byte RAM.

25.4 Text- und Grafikausgabe in ein Fenster

An der Überschrift können Sie schon erkennen, daß Intuition wenig Unterschied zwischen Text- und Grafikbehandlung macht. Ein printf-Aufruf führt bei Intuition-Fenstern leider nicht zum Erfolg. Das wird hier anders geregelt.

25.4.1 Text

Es gibt zur Stringausgabe eine Funktion namens Text. Ihr wird - und jetzt nicht nervös werden - ein Zeiger auf eine RastPort-Struktur übergeben, der in der Window-Struktur zu finden ist. Sie brauchen wirklich nicht zu verstehen, wie diese RastPort-Struktur aussieht oder welche Funktion sie zu erfüllen hat. Es genügt, wenn Sie den Ausdruck:

```
Window->RPort
```

den dafür zuständigen Routinen übergeben. Jetzt aber zurück zur Text-Funktion. Als weitere Parameter benötigt sie natürlich eine Zeichenkette und deren Länge. Das Format lautet also:

```
Text(Window->RPort, string, laenge);
```

25.4.2 Move

Vielleicht vermissen Sie die Angabe, wo denn der Text ausgegeben werden soll. Der String wird immer an der aktuellen Zeichenposition geschrieben. Diese wiederum wird mit der Funktion:

```
Move(Window->RPort, xpos, ypos);
```

gesetzt. Vor jedem Aufruf der Text-Routine steht dann in der Regel die Positionierung mittels "Move". Man kann und sollte sich deshalb eine eigene kleine Routine schreiben:

```
text(w_ptr, s, x, y)
struct Window *w_ptr;
char *s;
int x, y;
{
   Move(w_ptr->RPort, x, y);
   Text(w_ptr->RPort, s, strlen(s));
}
```

Um die Funktion allgemein zu halten, wird ihr auch der Zeiger auf das Window, auf dem der Text erscheinen soll, übergeben. Dadurch ist es möglich, verschiedene Windows mit der gleichen Funktion zu bedienen. Ein Aufruf sieht dann so aus:

```
text(Window, "Achtung!", 25, 40);
```

Der Text erscheint an der Position (25/40), wenn - und das ist sehr wichtig - das Fenster dies zuläßt. Im Fenster kann so viel geschrieben werden, wie man möchte. Intuition paßt aber auf, daß keine anderen Fenster oder gar der Screen überpinselt werden. Sollte der Text nicht ganz auf dem Fenster darstellbar sein, wird eben nur bis zum Fensterrand geschrieben. Sie können dadurch sicher sein, daß Sie nicht unbeabsichtigt in fremden Fenstern herummalen. Da wir gerade beim Malen sind, natürlich kann der Amiga auch das.

25.4.3 Draw

Mittels der Draw-Funktion kann man beliebige Linien zeichnen. Schaut man sich dann die Parameter der Routine an:

```
Draw(Window->RPort, x, y);
```

so fällt auf, daß auch hier etwas fehlt. Um eine Linie zu ziehen, benötigt man zwei Punkte. Mit Draw wird die Gerade von der aktuellen Position bis zum Punkt (x/y) gezogen.

Die vorgestellten Routinen sind nicht Teil der Intuition-Library, sondern gehören zu "graphics.library". Auch diese Library muß zuerst geöffnet werden und liefert einen speziellen Zeiger zurück. Alles verläuft analog zur Behandlung der Intuition-Library.

Zum Zeichnen benötigt man normalerweise die Koordinaten der Maus. Die finden wir in MouseX und MouseY, zwei Elemente der Window-Struktur. Damit hätten wir dann alles zusammen, um etwas in ein Fenster zu schreiben.

Im folgenden Programm tauchen auch die alten Bekannten "argv" und "argc" wieder auf. Dadurch können Sie vom CLI auch andere als die voreingestellten Werte verwenden. Aufgerufen wird nach dem Format:

```
prg X-AUFL Y-AUFL BITPLANES

z.B.

malomat 640 256 3
```

Interessant ist dabei, daß die Auflösung eines Screens größer sein kann als die maximale Auflösung des Bildschirms. 640 * 256 Punkte können dargestellt werden, wählt man aber beispielsweise 800 Punkte in der Horizontalen, so kann man die darauf befindlichen Windows über den rechten Bildschirmrand hinausschieben. Gleiches gilt übrigens auch für die Vertikale.

25.4.4 Kleines Malprogramm

```
#include <exec/types.h>
#include <intuition/intuition.h>

extern LONG OpenLibrary();
extern struct Screen *OpenScreen();
extern struct Window *OpenWindow();


struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;

#define INTUITION_REV 0
#define GRAPHICS_REV 0


struct TextAttr Font =
{
   "topaz.font",
   TOPAZ_SIXTY,
   FS_NORMAL,
   FPF_ROMFONT,
};
```

```
UBYTE screentitel[81];

struct NewScreen NewScreen =
{
   0,0,
   640,  /* Breite */
   256,  /* Dt. Version 256 Zeilen */
   2,    /* 2 Bitplanes = 4 Farben */
   2,3,  /* Farbregister */
   HIRES,
   CUSTOMSCREEN,
   &Font,
   screentitel,
   NULL,
   NULL,
};

struct NewWindow NewWindow =
{
   20, 20,   /* X- und Y-Position */
   400, 200, /* Breite, H he */
   0, 1,     /* Farben (0-3) */
   CLOSEWINDOW,
   WINDOWCLOSE | SMART_REFRESH | ACTIVATE
      | WINDOWSIZING | SIZEBRIGHT | WINDOWDRAG | WINDOWDEPTH,
   NULL,
   NULL,
   "Mal-Fenster",
   NULL,    /* Hier kommt sp ter der Screen-Pointer rein */
   NULL,
   190, 50,
   640, 256,
   CUSTOMSCREEN
};


main(argc, argv)
int argc;
char *argv[];
{
   struct Screen *Screen;
   struct Window *Window;
   register char s[81];  /* Zwischenspeicher */
   int farben = 4;
   register int x, y, xalt, yalt;
```

```
    if((IntuitionBase = (struct IntuitionBase *)
       OpenLibrary("intuition.library", INTUITION_REV)) == NULL)
       exit(FALSE);
    if((GfxBase = (struct GfxBase *)
       OpenLibrary("graphics.library", GRAPHICS_REV)) == NULL)
       exit(FALSE);

    if(argc != 4)
       {
          printf("Fehlerhafte Argumente!\n");
          printf("X-Aufl Y-Aufl Bitplanes\n");
       }
    else
       {
          NewScreen.Width  = atoi(argv[1]);
          NewScreen.Height = atoi(argv[2]);
          NewScreen.Depth  = atoi(argv[3]);
          if(NewScreen.Depth > 4 || NewScreen.Depth < 1)
             NewScreen.Depth = 2;
          farben = 1 << NewScreen.Depth;
          NewScreen.DetailPen  = farben - 1;
          NewScreen.BlockPen   = farben - 2;
       }

    sprintf(screentitel, "Dieser HIRES-Screen hat %d Farben", far-
ben);

    if( (Screen = OpenScreen(&NewScreen) ) == NULL)
       exit(FALSE);
    NewWindow.Screen = Screen;

    if(argc == 4)
       {
          NewWindow.Width     =  Screen->Width  / 2;
          NewWindow.Height    =  Screen->Height / 3;
          NewWindow.MinWidth  =  Screen->Width  / 3;
          NewWindow.MinHeight =  Screen->Height / 5;
          NewWindow.MaxWidth  =  Screen->Width;
          NewWindow.MaxHeight =  Screen->Height;
       }

    if( (Window = OpenWindow(&NewWindow) ) == NULL)
       exit(FALSE);
    text(Window, "Hall chen!", 20, 20);
```

```
   /* Mit Startwert initialisieren */
   Move(Window->RPort, xalt = Window->MouseX, yalt = Window-
>MouseY);

   /* Hier wird gemalt bis man an den linken oder oberen Rand
kommt */
   while( (x = Window->MouseX) > 0 && (y = Window->MouseY) > 0)
      {
         sprintf(s, "X =%3d, Y =%3d", x, y);
         text(Window, s, 150, 7);
         Move(Window->RPort, xalt, yalt);
         Draw(Window->RPort, xalt = x, yalt = y);
      }

   text(Window, " Bitte Close-Gadget ", 20, 20);
   /* Wieder auf das Close-Gadget warten! */
   Wait(1 << Window->UserPort->mp_SigBit);

   CloseWindow(Window);          /* Aufr umen! */
   CloseScreen(Screen);
   CloseLibrary(GfxBase);
   CloseLibrary(IntuitionBase);
   exit(TRUE);
}


text(w_ptr, s, x, y)
struct Window *w_ptr; /* Window, das beschrieben werden soll */
char *s;              /* Auszugebender Text */
int x , y;            /* Koordinaten */
{
   Move(w_ptr->RPort, x, y);
   Text(w_ptr->RPort, s, strlen(s));
}
```

Öfter mal was Neues, das ist auch hier die Devise. Zuerst fällt die neue Struktur TextAttr mit der Variablen Font auf. Jede NewScreen-Struktur besitzt unter anderem eine Komponente namens "Font". Hier kann man den zu verwendenden Zeichensatz (Adresse der Struktur) eintragen. Natürlich ist dies auch wieder ein Zeiger auf eine weitere Struktur, eben "TextAttr".

Die Struktur zur Definition eines Zeichensatzes ist recht einfach. Zuerst wird der Name des Fonts aufgeführt, dann die Höhe der Zeichen und die Darstellungsart. Zuletzt wird durch ein Flag die Stelle beschrieben, an der sich der Zeichensatz befindet. Wir benutzen den eingebauten Zeichensatz, der sich im ROM befindet. Außerdem wird dieser Font in der 60-Zeichen-Version benutzt. Er ist etwas größer als der sonst genutzte mit 80 Zeichen pro Zeile. Dafür muß man TOPAZ_SIXTY durch TOPAS_EIGHTY ersetzen.

Zwar sind alle Einträge in NewWindow und NewScreen bereits vorbelegt, sollte jedoch der Benutzer andere Werte für Auflösung und Bitplanes wünschen, so werden diese übernommen. Die Maximal- und Minimalgrößen des Windows müssen ebenfalls angepaßt werden. Da die Farbindices des Windows mit 0 und 1 immer einsatzfähig sind, braucht man hier keine Umrechnung vorzunehmen. Der Screen hingegen erhält die letzten beiden Farbregister, die natürlich von der Anzahl der Farben abhängen. Maximal läßt das Programm 16 Farben gleich 4 Bitplanes im hochauflösenden Modus zu. Der Titel des Screens wird danach mit dieser Information versehen. Deshalb muß auch eine zusätzliche Variable verwendet werden, da in der Struktur kein Speicher für den Titel vorgesehen ist.

Sobald das Fenster geöffnet ist, wird auch der erste Text ausgegeben. Dann kann man mit der Maus ein wenig herummalen. Solange der Mauszeiger nicht oben oder links aus dem Fenster bewegt wird, folgt ihm eine durchgezogene Linie. Außerdem wird in der Titelzeile stets die Mausposition angegeben. Diese ist nicht absolut, sondern relativ zur linken oberen Ecke des Windows. Es sind dadurch auch negative Werte möglich, wenn die Maus über den linken oder oberen Rand geschoben wird. Dies wird vom Programm als Abbruchkriterium verwendet.

Durch die zwischenzeitliche Ausgabe der Maus-Koordinaten muß die aktuelle Zeichenposition in xalt und yalt zwischengespeichert werden. Vor dem Zeichnen der Linie werden diese Werte dann mit der Move-Funktion wiederhergestellt. Soviel zu diesem Programm.

Nun noch einige Anregungen. Bislang benutzten wir bei selbstdefinierten Screens stets das Define HIRES in "ViewModes", wodurch in der Horizontalen eine Auflösung bis zu 640 Punkte erzielt werden kann. Der Amiga kann aber auch mit einer niedrigen Auflösung aufwarten, die mit 320 Punkten eine ganze Bildschirmzeile füllt. Bei gleicher Farbanzahl benötigt sie nur die Hälfte an Speicherplatz. Außerdem wird hierdurch erst die Möglichkeit eröffnet, mit 5 Bitplanes und dadurch mit 32 verschiedenen Farben zu hantieren. Das einzige, was Sie dazu tun müssen, ist das HIRES durch NULL zu ersetzen. Probieren Sie es doch mit unserem ersten Screen-Programm gleich aus. Außer obiger Änderung muß nur noch die Breite des Screens auf die geforderten 320 Pixel gesetzt werden. Nun kann, vorausgesetzt es sind 5 Bitplanes angefordert, aus einem Topf von 32 Farben mit den Farbregistern 0 - 31 geschöpft werden.

25.4.5 Niedrige Auflösung und Interlace

Vielleicht sagt Ihnen der Interlace-Modus etwas. Dieser Modus kann ebenfalls in "ViewModes" eingeschaltet werden. Durch ihn erhält das Bild eine vertikale Auflösung von 512 (deutsche Version) statt 256 Punkten. "Nicht schlecht!", würde man vermuten, doch kommt nun das große "aber". Dieser Vorgang ist nur dadurch realisierbar, daß die Bildwiederhol-Frequenz des Monitors von 50 auf 25 Hz gesenkt wird. Im Gegensatz zum Fernseher entsteht ein sehr stark flimmerndes Bild, das man auf einem Amiga-Monitor wohl nicht lange aushalten kann. Der Modus ist aber auch nur für die Monitore gedacht, die eine lange Nachleuchtdauer haben und dadurch keinerlei Flackern zeigen. Aber schauen Sie sich dieses Bild doch ruhig einmal selbst an und bilden Ihr eigenes Urteil. Das einzige, was zu tun ist, ist das Define "LACE" hinzuzufügen. Ein paar Beispiele:

```
Niedrigauflösend (320 Punkte):
NewScreen.ViewModes = NULL;

Hochauflösend (640 Punkte) und Interlace (512 Punkte):
NewScreen.ViewModes = HIRES | LACE;
```

Niedrigauflösend (320 Punkte) und Interlace (512 Punkte):
NewScreen.ViewModes = LACE;

Bei nunmehr 32 Farben, die Sie nutzen können, wäre eine Funktion sicher hilfreich, die es uns erlaubt, die Stiftfarbe zu wechseln. Hierfür steht:

```
SetAPen(Window->RPort, color);
```

bereit, mit der sich die aktuelle Farbe auf das Farbregister "color" einstellen läßt. Auch um diese Routine in Aktion zu sehen, muß kein neues Programm geschrieben werden. Nehmen wir unser kleines Malprogramm und fügen folgende Zeilen ein:

Zu Beginn der main-Funktion die Definition einer zusätzlichen Variable namens "color":

```
register int x, y, xalt, yalt, color = 1;
```

Am Ende der while-Schleife noch 2 Zeilen, hier der Ausschnitt:

```
while( (x = Window->MouseX) > 0 && (y = Window->MouseY) > 0)
   {
      sprintf(s, "X =%3d, Y = %3d", x, y);
      text(Window, s, 150, 7);
      Move(Window->RPort, xalt, yalt);
      Draw(Window->RPort, xalt = x, yalt = y);

      SetAPen(Window->RPort, color++);  /* Neu! */
      if(color == farben) color = 1;    /* Neu! */
   }
```

Zu Beginn wird die Variable "color" auf 1 gesetzt, damit mit dem ersten Farbindex gezeichnet werden kann (0 ist ja die Hintergrundfarbe). Nach jedem kleinen Linienfragment wechselt nun die Stiftfarbe, bis der letzte Index (Anzahl der Farben - 1) erreicht ist. Dann geht es wieder von vorne los. So sehen Sie (vielleicht) auch zum ersten Mal alle Farben, inklusive der Register, die wir sonst nicht benutzt haben. Wenn es noch bunter als die 16 im HIRES-Modus möglichen Farben werden soll, ändern Sie die Sicherheitsabfrage:

```
if(NewScreen.Depth > 4 || NewScreen.Depth < 1)
     NewScreen.Depth = 2;
```

in

```
if(NewScreen.Depth > 5 || NewScreen.Depth < 1)
     NewScreen.Depth = 2;
```

und ändern Sie wie zuvor beschrieben HIRES in NULL. Beachten Sie wieder die verringerte X-Auflösung von 320 anstelle der 640 Punkte. Schon können Sie mit bis zu 32 Farben auf einem Window herumzeichnen.

Da die vielen Farben und die hohe Auflösung mit Speicherplatz bezahlt werden, hier mal einige Beispiele, was Windows oder Screens so verschlingen können:

Niedrige Auflösung, Interlace-Modus, 32 Farben
320 (Pixel) * 512 (Pixel) * 5 (Bitplanes) / 8 (Bits pro Byte)
= 102.400 Byte = 100 K Byte

Hohe Auflösung, Interlace-Modus, 8 Farben
640 (Pixel) * 512 (Pixel) * 3 (Bitplanes) / 8 (Bits pro Byte)
= 122.880 Byte = 120 K Byte

Niedrige Auflösung, 2 Farben
320 (Pixel) * 256 (Pixel) * 1 (Bitplane) / 8 (Bits pro Byte)
= 10.240 Byte = 10 K Byte

Die Unterschiede sind gewaltig! Denken Sie daran, wenn Ihr Rechner (noch) kein Mega-Amiga ist.

25.4.6 Pixelbearbeitung

Weitere Grafikbefehle neben Draw sind:

```
ReadPixel(Window->RPort, x, y);
```

und

```
WritePixel(Window->RPort, x, y);
```

Deren Funktionen sind aus dem Namen bereits klar ersichtlich. ReadPixel prüft, ob an der angegebenen Position ein Punkt gesetzt ist und liefert den Farbwert des Pixels zurück. Sollte kein Punkt sichtbar sein, weil der Punkt die Hintergrundfarbe besitzt, so erhält man ebenfalls die entsprechende Registernummer, in diesem Fall 0. Sollte der Punkt außerhalb des Windows liegen, dessen Rastport übergeben wurde, ist das Resultat -1.

Das Gegenteil von ReadPixel ist WritePixel. Diese Funktion setzt lediglich einen einzelnen Punkt an der übergebenen Position. Die dafür verwendete Farbe richtet sich wie bei allen anderen Routinen nach der aktuellen Zeichenfarbe, die ja mit SetAPen bestimmt wird.

Im folgenden ein Programm, das mit diesen beiden Befehlen die Titelzeile des benutzten Windows sinusartig verformt.

```
#include <exec/types.h>
#include <intuition/intuition.h>

extern struct Window *OpenWindow(); /* Saubere Deklaration */
extern long *OpenLibrary();         /* Hallo Aztek-User! */
extern double sin();


struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;


#define INTUITION_REV 0
#define GRAPHICS_REV 0

/* Farbanzahl der Workbench */
#define WB_FARBEN 4


struct NewWindow NewWindow =
{
   10, 50,   /* X- und Y-Position */
   360, 120, /* Breite, H he */
   3, 2,     /* Farbindices */
```

```
   NULL,
   SMART_REFRESH | ACTIVATE | WINDOWDRAG | WINDOWDEPTH,
   NULL,
   NULL,
   "DIESE ZEILE WIRD SINUSARTIG VERBOGEN!",
   NULL,
   NULL,
   0, 0,
   640, 256,
   WBENCHSCREEN
};



main()
{
   struct Window *Window;
   register struct RastPort *r;
   register int i, j, top, yoffset;
   int i_bis, j_bis, color, farben[512];
   double faktor;

   if((IntuitionBase = (struct IntuitionBase *)
      OpenLibrary("intuition.library", INTUITION_REV)) == NULL)
      exit(FALSE);

   if((GfxBase = (struct GfxBase *)
      OpenLibrary("graphics.library", GRAPHICS_REV)) == NULL)
      exit(FALSE);


   if((Window = OpenWindow(&NewWindow)) == NULL)
      exit(FALSE);

   r = Window->RPort;
   top = Window->Height / 4;
   faktor = 2 * 3.1415926 / Window->Width * 1.5;  /* 1.5 Sinuswel-
len */

   for(i = 2, i_bis = Window->Width - 2; i < i_bis; i++)
      {
         for(j = 0; j < top; j++)    /* eine komplette vertikale
Zeile */
            {                        /* in das Array ³bertragen */
               color = ReadPixel(r, i, j);
               if(++color == WB_FARBEN)
```

```
                    farben[j] = 0;
                 else
                    farben[j] = color; /* und Farbindex um eins
erh hen */
              }
           for(j = 0,
              yoffset = top + top * sin(faktor * i) + 16;
              j < top; j++)
              if(farben[j]) /* Falls Punkt gesetzt werden soll */
                 {
                    SetAPen(r, farben[j]);
                    WritePixel(r, i, j + yoffset);
                 }
        }
     Delay(1500);       /* Warte 1500 Ticks = 30 sec */
     CloseWindow(Window);
     CloseLibrary(GfxBase);
     CloseLibrary(IntuitionBase);
     exit(TRUE);
  }
```

Erläuterungen zum Programm:

Das Programm läuft unter dem Workbench-Screen und benutzt dessen Farben. Es werden dabei 2 Bitplanes, also 4 Farben vorausgesetzt. Sollten Sie eine davon abweichende Workbench aufgebaut haben, muß das Define WB_FARBEN entsprechend angepaßt werden. Das Window, das in der NewWindow-Struktur definiert wird, enthält lediglich die Gadgets, um das Fenster vor oder hinter andere zu verschieben. Die Größe kann nicht geändert werden.

In einer großen Schleife, die die gesamte Fensterbreite abläuft, werden zuerst alle zu einer X-Position gehörenden Pixel in einem Array gesammelt. Bevor allerdings der von ReadPixel zurückgegebene Wert abgespeichert wird, erfolgt erst noch eine Farbtransformation. Jedes Pixel erhält die Farbe aus dem folgenden Register. Der zu übertragende Bereich des Bildschirms ist das obere Viertel des Windows. Damit sich die zu lesenden und zu schreibenden Informationen nicht in die Quere kommen, wird erst eine ganze Spalte in das Array "farben" gerettet. Dann wird mittels der Sinusfunktion die neue Position der Punkte er-

rechnet und dort mit WritePixel in der neuen Farbe plaziert. Man kann allerdings die Pixel ausnehmen, die die Hintergrundfarbe (Farbindex 0) erhalten. Deshalb wird dies auch zuerst geprüft, um etwas Geschwindigkeit herauszuholen.

Nach vollständiger Transformation erfolgt der Aufruf der Delay-Funktion, die das Programmende etwas hinauszögert und Ihnen die Möglichkeit gibt, das Window noch einen Moment zu betrachten. Der Parameter dazu gibt die Wartezeit in 1/50 sec (Ticks) an. Um 30 Sekunden Verzögerung einzubauen, ist demnach 1500 zu übergeben.

25.5 DOS

Neben Intuition ist auch das DOS bei der Programmierung nicht ganz uninteressant. Im Gegenteil, viele Programme würden ohne Unterstützung durch das Amiga-DOS ziemlich dumm dastehen. So übernehmen DOS-Routinen nicht nur das Löschen und Umbenennen von Dateien, auch das Erstellen von Directories oder das Auslesen eines Inhaltsverzeichnis werden hierdurch gemanagt.

Auch das DOS ist in einer Library ("dos.library") zusammengefaßt. Im Gegenteil zu anderen Libraries ist diese aber für uns stets geöffnet. Wir brauchen sie also weder zu öffnen noch zu schließen. Die Benutzung ist genauso einfach, wie Sie es von Funktionen aus den Standardbibliotheken gewöhnt sind. Als Beispiel den Aufruf zum Löschen einer Datei:

```
erfolg = DeleteFile(filename);
```

Als Rückgabewert erhält man eine Integer-Zahl, die durch null einen Fehler, ungleich null die korrekte Verarbeitung anzeigt. Mit dieser Funktion können aber nicht nur einzelne Dateien, sondern auch Directories eliminiert werden. Es darf sich dann allerdings keine Datei mehr darin befinden, die müssen zuvor gelöscht werden. Ebenso einfach benutzt man die Funktion Rename zum Umbennen von Dateien:

```
erfolg = Rename(alter_name, neuer_name);
```

Die Erfolgsmeldung wird auch hier durch einen Wert ungleich null angezeigt. Die anderen Parameter "alter_name" und "neuer_name" sind wie auch obiger "filename" Strings, die einen gültigen Dateinamen enthalten.

25.6 SetComment

Viel interessanter als die oben genannten Funktionen ist aber eine Routine, die es erlaubt, einen Kommentar an eine Datei anzuhängen. Diese Bemerkung ist völlig unabhängig vom Inhalt und der Größe der Datei und wird dort untergebracht, wo auch der Dateiname und deren Parameter stehen. Das DOS-Kommando FILENOTE im C-Ordner kann einen solchen Text an eine bestehende Datei "anhängen". Wir können dies ebenfalls mit der Funktion:

```
erfolg = SetComment(filename, kommentar);
```

erledigen. Ein kurzes Programm zeigt die Routine in Aktion.

```
#include <libraries/dos.h>


main(argc, argv)
int argc;
char *argv[];
{
   if(argc == 3)
      {
         if(!SetComment(argv[1], argv[2]))
            printf("Fehler %d\n", IoErr());
      }
   else
      printf("Format: MAKECOM DATEI KOMMENTAR\n");

   exit(TRUE);
}
```

Über die Kommandozeile werden dem Programm der Dateiname und der zu übertragene Kommentar übermittelt. Wie bei allen Datentransfers in dieser Weise ist die Verwendung von Leerzeichen unzulässig.

25.7 Inhaltsverzeichnis auslesen

Eine wichtige Aufgabe für Programme, die mit Dateien hantieren, ist das Auslesen des Inhaltsverzeichnisses. Um dies im Programm zu realisieren, sind einige Funktionen notwendig.

Zuerst kommt die Routine Lock an die Reihe. Lock, durch Schloß oder schließen zu übersetzen, fixiert ein bestimmtes Directory. Erst danach kann man mit anderen Funktionen in diesem Verzeichnis herumwerkeln. Lock wird der Name des Verzeichnisses und der Zugriffsmodus (lesen oder schreiben) übergeben. Letzterer ist ein Integer-Wert, den man am besten mit dem dafür vorgesehenen Define ACCESS_READ (zum Lesen) verwendet. Zurückgegeben wird ein Schlüssel, mit dem man, vergleichbar mit dem Handle bei Dateizugriffen oder dem Windowpointer unter Intuituion, nun stets dieses eine Verzeichnis bearbeiten kann. Sollte dieser allerdings null sein, so ist ein Fehler aufgetreten.

Zum Auslesen des Inhaltsverzeichnisses benötigt man zwei Funktionen. Die eine ist Examine, die andere ExNext. Zuerst muß Examine aufgerufen werden, um den ersten Eintrag des Directories zu erhalten. Danach muß für jedes weitere File oder Directory ein Aufruf von ExNext folgen. Beide Routinen erfordern nicht nur den ermittelten Lock, sondern auch einen Zeiger auf die FileInfoBlock-Struktur. In ihr werden alle wichtigen, eine Datei betreffende Daten abgelegt.

```
erfolg = Examine(lock, &fileinfo);
```

Und hier die Strukturdefinition:

```
struct FileInfoBlock {
LONG fib_DiskKey;
LONG fib_DirEntryType;
```

```
char fib_FileName[108];
LONG fib_Protection;
LONG fib_EntryType;
LONG fib_Size;
LONG fib_NumBlocks;
struct DateStamp fib_Date;
char fib_Comment[116];
}
```

Wichtig für uns sind folgende Informationen:

- *fib_DirEntryType*

  zeigt an, ob es sich um eine normale Datei (<0) oder um ein Directory (>0) handelt.

- *fib_FileName*

  enthält den Namen, der max. 30 Zeichen lang ist, auch wenn hier großzügig auf 108 Zeichen definiert wurde.

- *fib_Protection*

  enthält Flags, die anzeigen, ob man die Datei lesen, beschreiben, ausführen oder löschen darf. Beschäftigen wir uns damit etwas näher. Die Variable ist zwar als LONG definiert (=32 Bit), davon werden jedoch nur die untersten 4 Bit benötigt. Die Prioritäten liegen in der Reihenfolge vor:

```
R W E D
8 4 2 1

R = Read (lesen)
W = Write (schreiben)
E = Execute (ausführen)
D = Delete (löschen)
```

Für jeden Schutz vor einem der oben aufgeführten Aktionen muß das entsprechende Bit gesetzt werden. Darf beispielsweise eine Datei oder Directory nur gelesen oder gelöscht werden, so müssen die Flags W und E gesetzt werden.

```
RWED
0110 (Bits) = 4 + 2 = 6
```

In dieser Variable muß also der Wert 6 enthalten sein. Wichtig ist, immer die Flags zu setzen, deren Funktion man verbieten möchte. Wenn Sie selbst Änderungen an diesen Informationen vornehmen wollen, bedienen Sie sich bitte des Programms PROTECT von der Workbench. Mit dem Programm werden die Flags, die Sie übergeben, so umgewandelt, daß deren Funktion ausführbar ist. Dies ist genau das Gegenteil von dem, was wir im eigenen Programm verwenden müßten. Um eine Datei vor dem Löschen zu schützen, wäre die folgende Zeile notwendig:

```
PROTECT ED RWE
```

- *fib_Size*

  Dateigröße in Bytes

- *fib_NumBlocks*

  enthält die Zahl der belegten Blöcke auf der Diskette.

- *fib_Date*

  Datum beim letzten Beschreiben des Files; ist in einer separaten Struktur untergebracht.

- *fib_Comment*

  ist der schon bekannte Kommentar zu der Datei

Zum Auslesen haben wir alle nötigen Daten zusammen, so daß es nicht mehr schwierig ist, daraus das endgültige Programm zu stricken. In der folgenden Version benutzen wir wieder der Einfachheit halber die Parameterübergabe mittels der Kommandozeile. Dieser Parameter gibt das Verzeichnis an, das ausgelesen werden soll.

```
#include <libraries/dos.h>

struct FileInfoBlock fi;


main(argc, argv)
int argc;
char *argv[];
{
   long lock;
   int error;
   char filepath[100];

   if(argc == 2)    /* Parameter vorhanden? */
      strcpy(filepath, argv[1]);
   else
      strcpy(filepath, "sys:");

   lock = Lock(filepath, ACCESS_READ);
   printf("Lock-Wert %d\n", lock);
   if(!lock)
      {
         printf("Kein Lock! ERROR!\n");
         exit(FALSE);
      }

   if(Examine(lock, &fi))  /* Erster Aufruf erfolgreich? */
      do
         ausgabe(); /* R³ckgabewert interessiert momentan nicht */
      while(ExNext(lock, &fi));  /* solange, bis Fehler auftritt
*/

   error = IoErr();  /* Welcher Fehler? */
   if(error != ERROR_NO_MORE_ENTRIES) /* Ein "richtiger" Fehler!
*/
      printf("Fehler %d aufgetreten!\n", error);

   exit(TRUE);
}


ausgabe()
{
   if(!*fi.fib_FileName)  /* strlen = 0 */
      {
```

```
            printf("Leer!\n");   /* z.B. Hauptdirectory der RAM-Disk
*/
            return(0);           /* daher Directory ohne Namen */
         }
      if(fi.fib_DirEntryType > 0)
         printf("Directory-Name");
      else
         printf("Dateiname     ");

      printf(": >%20s< RWXD %lx Bytes: %-6ld Blocks %-4ld\n",
         fi.fib_FileName, fi.fib_Protection, fi.fib_Size,
   fi.fib_NumBlocks);

      if(*fi.fib_Comment)  /* Falls Kommentar vorhanden, ... ausgeben
*/
         printf("Kommentar: >%s<\n", fi.fib_Comment);
      return(fi.fib_DirEntryType > 0);   /* File-Typ zur³ck */
   }
```

Die Protect-Flags werden in unserem Programm nicht gesondert aufgeschlüsselt (wäre auch kein Problem), sondern als Hexzahl dargestellt ("%lx"-Formatzeichen). Das Programm entsprechend zu erweitern, ist doch mal eine Aufgabe an Ihre Programmierkunst.

Mit dem DOS-Befehl LIST von der Workbench können Sie sich die Flags in der geeigneten Weise, in "RWED" gesplittet, anzeigen lassen. Damit ist genau das richtige Team zusammen. Mit PROTECT setzen Sie beliebige Flags, mit LIST sehen Sie die Auswirkungen und können diese mit den Informationen vergleichen, die das eigene Programm liefert. So, jetzt sind Sie an der Reihe! Es gibt viel zu tun, packen Sie's an. Viel Spaß!

Anhang A: Funktionen

Dateiname: "strlen.c"

```
/*********************************************************************/
/* Name:          strlen                                             */
/* Parameter:     s (String)                                         */
/* Rückgabewerte: Laenge (int)                                       */
/* Funktion:      Anzahl der Zeichen in "s"                          */
/* Sonstiges:     -                                                  */
/*********************************************************************/

strlen(s)
char s[];
{
  register int i = 0;
  while(s[i])
    i++;
  return(i);
}
```

Dateiname: "strcpy.c"

```
/*********************************************************************/
/* Name:          strcpy                                             */
/* Parameter:     s (String), t (String)                             */
/* Rückgabewerte: -                                                  */
/* Funktion:      Kopiert "s" nach "t"                               */
/* Sonstiges:     -                                                  */
/*********************************************************************/

strcpy(t,s)
register char *t,*s;
{
  while(*t++ = *s++)
    ;
}
```

Dateiname: "strcat.c"

```
/*********************************************************************/
/* Name:          strcat                                             */
/* Parameter:     s (String), t (String)                             */
/* Rückgabewerte: -                                                  */
/* Funktion:      "t" an "s" anhängen                                */
/* Sonstiges:     -                                                  */
/*********************************************************************/

strcat(s,t)
register char *s,*t;
{
  while(*s)
    s++;
  while(*s++ = *t++);
}
```

Dateiname: "buchstab.c"

```
/*********************************************************************/
/* Name:          buchstab                                           */
/* Parameter:     z (char)                                           */
/* Rückgabewerte: Es war ein Buchstabe (1), sonst (0)                */
/* Funktion:      Bestimmt, ob Buchstabe oder nicht                  */
/* Sonstiges:     -                                                  */
/*********************************************************************/
#define FALSE 0
#define TRUE  1

buchstab(z)
register char z;
{
  if ((z >= 'a' && z <= 'z') || (z >= 'A' && z <= 'Z')) return(TRUE);
  return(FALSE);
}
```

Dateiname: "z_comp.c"

```
/*********************************************************************/
/* Name:          z_comp                                             */
/* Parameter:     z1 (char), z2 (char)                               */
/* Rückgabewerte: 1 (gleich), 0 (ungleich)                           */
/* Funktion:      Vergleicht 2 Zeichen                               */
/* Sonstiges:     benötigt buchstabe()                               */
/*********************************************************************/
#define FALSE 0
#define TRUE  1
extern int grklflag;

z_comp(z1,z2)
register char z1,z2;
{
  if (z1 == z2) return(TRUE);
  if( grklflag && buchstab(z1) && buchstab(z2))
    if((z1 + 'a' - 'A' == z2)||(z2 + 'a' - 'A' == z1)) return(TRUE);
  return(FALSE);
}
```

Dateiname: "strcmp.c"

```
/*********************************************************************/
/* Name:          strcmp                                             */
/* Parameter:     s (String), t (String)                             */
/* Rückgabewerte: Gleich 0  Ungleich 1                               */
/* Funktion:      Vergleicht "s" und "t"                             */
/* Sonstiges:     -                                                  */
/*********************************************************************/

strcmp(s,t)
register char *s, *t;
{
  register int gleich;
  while(gleich = z_comp(*s, *t++) )
    if(!*s++)
      return(0);
  return(!gleich);
}
```

Dateiname: "strchar.c"

```
/*********************************************************************/
/* Name:          strchar                                            */
/* Parameter:     s (String), c (char)                               */
/* Rückgabewerte: Position (int), oder -1                            */
/* Funktion:      Ermittelt Position des Zeichen "c" in "s"          */
/* Sonstiges:     -                                                  */
/*********************************************************************/

strchar(s,c)
register char s[];
register char c;
{
  register int i = 0;
  while(!z_comp(s[i],c) && s[i])
    i++;
  if(s[i]) return(i);
  return(-1);
}
```

Dateiname: "strchbac.c"

```
/*********************************************************************/
/* Name:          strchback                                          */
/* Parameter:     s (String), c (char)                               */
/* Rückgabewerte: Index (int)                                        */
/* Funktion:      Sucht Position des Zeichen "c" in "s"              */
/* Sonstiges:     benötigt z_comp(), strlen()                        */
/*********************************************************************/

strchback(s,c)
register char s[];
register char c;
{
  register int i = strlen(s);
  while((i >= 0) && !z_comp(s[i],c) )
    i--;
  return(i);  /* Fehler = -1 */
}
```

Dateiname: "ilatoila.c"

```
/*********************************************************************/
/* Name:          ltoa                                               */
/* Parameter:     n (long), s (String)                               */
/* Rückgabewerte: -                                                  */
/* Funktion:      Wandelt Longwert in Zeichenkette um                */
/* Sonstiges:     benötigt reverse()                                 */
/*********************************************************************/
#define TRUE 1
#define EOS '\0'

ltoa(n, s)
register char s[];
register long n;
{
  register int i = 0;
  register int vorzeich = 0;
  if (n < 0)
    {
      vorzeich = TRUE;
      n = -n;
    }
  do
    {
      s[i++] = n % 10 + '0';
    } while((n /= 10) > 0);
  if(vorzeich)
    s[i++] = '-';
  s[i] = EOS;
  reverse(s);
}


/*********************************************************************/
/* Name:          itoa                                               */
/* Parameter:     n (int), s (String)                                */
/* Rückgabewerte: -                                                  */
/* Funktion:      Wandelt Integerzahl in Zeichenkette um             */
/* Sonstiges:     benötigt ltoa()                                    */
/*********************************************************************/

itoa(n, s)
register int n;
```

```
register char s[];
{
  ltoa((long)(n),s);
}


/*********************************************************************/
/* Name:          atol                                               */
/* Parameter:     s (String)                                         */
/* Rückgabewerte: n (long)                                           */
/* Funktion:      Wandelt Zeichenkette in Longwert um                */
/* Sonstiges:     -                                                  */
/*********************************************************************/

long atol(s)
register char *s;
{
  register long val;
  register int sign = 1;
  while(*s == ' ')
    s++;
  if (*s == '+' || *s == '-')
    sign = (*s++ == '+') ? 1 : -1;
  for(val = 0; *s >= '0' && *s <= '9'; ++s)
    val = 10 * val + *s - '0';
  return(sign * val);
}


/*********************************************************************/
/* Name:          atoi                                               */
/* Parameter:     s (String)                                         */
/* Rückgabewerte: Integerzahl                                        */
/* Funktion:      Wandelt Zeichenkette in Integer um                 */
/* Sonstiges:     -                                                  */
/*********************************************************************/

atoi(s)
register char *s;
{
  long atol();
  return(atol(s));
}
```

Anhang B: Die Geschichte von C

C wurde bereits Mitte der siebziger Jahre von einem einzigen Mann entwickelt: Dennis M. Ritchie, der zu dieser Zeit bei der amerikanischen Firma Bell Laboratories arbeitete. C stammt von dem Vorgänger, der Sprache B, ab. Sie sehen, woher diese Sprache ihren einfallsreichen Namen hat. Auch B wurde von einer anderen Sprache aus entwickelt: BCPL (Basic Cambridge Programming Language), was so viel wie Elementare Programmiersprache aus Cambridge heißt. C war ursprünglich für die Entwicklung eines Betriebssystems gedacht, das unter anderem Multi-User- und Multi-Tasking-fähig sein sollte, nämlich UNIX. Daraus erklärt sich, warum C-Programme so schnell sind. Multi-Tasking-Prozesse verlangen nach einem sehr schnellen Betriebssystem, das bis dato nur in Assembler geschrieben werden konnte. Daher entwickelte D. Ritchie die Sprache C, um die recht fehleranfällige und unübersichtliche Assemblerprogrammierung zu umgehen. Das Resultat, das Betriebssystem UNIX, besteht aus ca. 13000 Zeilen, wovon nur ein minimaler Teil von ca. 800 Zeilen für zeitkritische Routinen in Assembler geschrieben wurde. Der Rest ist C. Populär wurde C eigentlich erst durch Einführung des Amiga und des Atari ST, die C ebenfalls als Programmiersprache ihres Betriebssystems verwenden. So ist die Benutzeroberfläche Intuition des Amiga fast vollständig in C geschrieben. Die Profis und Softwarehäuser benutzen deshalb vorzugsweise C zur Entwicklung neuer Programmprojekte. C hat nämlich noch einen weiteren Vorteil: C ist portabel. Das heißt nichts anderes, als daß man C-Programme (in der Theorie) fast unverändert auf alle Rechner übertragen kann. Dies liegt daran, daß C nur eine geringe Anzahl von Befehlen aufweist, die bei allen Compilern vorhanden sind. Rechnerspezifische Teile beispielsweise für Ein- und Ausgabe gehören nicht zum eigentlichen Sprachumfang. Diese Routinen werden in sogenannten Bibliotheken mitgeliefert, die auf die jeweiligen Eigenarten des Rechners zugeschnitten sind. Der C-Programmierer braucht sich darum nicht zu kümmern. Er weiß, daß z.B. die Funktion getchar ein Zeichen von der Tastatur holt, egal ob das Programm auf einem C64, einem IBM PC, einem Amiga oder einem Atari ST laufen soll. Für die Softwarehäuser bedeutet diese Portabilität natürlich einen geringeren

Programmieraufwand, wenn ein Programm auf verschiedene Rechner übertragen werden soll.

Aufbau eines C-Compilers

Jeder C-Compiler ist in verschiedene Programmteile gesplittet, die je nach Hersteller entweder in einem Programm-Modul oder in mehreren Teilprogrammen vorliegen.

Der erste Teil, der seine Arbeit an einem C-Programm beginnt, ist der Makro-Präprozessor, der lediglich einzelne Textteile durch andere ersetzt (übrigens ganz nach unseren Wünschen). Das Ergebnis seiner Bemühungen ist aber immer noch eine reine Textdatei, die wir ohne weiteres mit dem Editor bearbeiten könnten. Dieses Resultat wird dem Scanner übergeben, der nach den C-spezifischen Befehlsworten sucht, diese erkennt und in einem Kurzformat abspeichert. Bei diesem Format wird der Befehl (z.B. "continue") nicht als 8 einzelne Buchstaben, sondern als Kennzahl (Token) codiert, die dadurch viel weniger Speicherplatz beansprucht. Zudem wird durch das "Tokenisieren" die weitere Übersetzung beschleunigt.

Nach Beendigung dieses Testlaufes kommt der Parser an die Reihe. Er prüft die Syntax der Befehle im Text und unterscheidet zwischen richtigen und falschen Zusammenstellungen von C-Befehlen. Ihm sind alle Regeln bekannt, wie Ausdrücke miteinander verknüpft werden können. Ähnlich wie in der Umgangssprache ist es mit dem Aneinanderhängen von Worten nicht getan, sondern es muß ein Sinn enthalten sein. Und diesen Sinn prüft der Parser.

Als wirklich letzten Teil des eigentlichen C-Compilers wird der Codegenerator in die Schlacht geworfen. Wie der Name schon andeutet, wandelt er den vom Parser bearbeiteten Text in entsprechende Maschinenbefehle um. Einige C-Compiler übersetzen die Maschinenbefehle erst noch in Assembler, so daß es dem Programmierer möglich ist, noch letzte Hand an den erzeugten Code zu legen. Das ist aber meistens überhaupt nicht mehr nötig, da die derzeitigen C-Compiler bereits sehr effizienten

Maschinencode produzieren. Nach Abschluß dieses Laufes befindet sich dann die Objektdatei mit der Endung ".o" auf einer Ihrer Disketten.

Anhang C: Der Lattice-C-Compiler

Als Voraussetzungen neben dem Compiler selbst wird vom Hersteller ein Laufwerk und eine Harddisk empfohlen. Der Compiler läuft aber auch mit "nur" zwei Laufwerken. Natürlich ist die erste Möglichkeit die beste, leider aber auch die teuerste. Die zweite ist schon realistischer, wenngleich auch nicht jeder zwei Laufwerke besitzt. Wenn Sie allerdings nur mit einem Laufwerk arbeiten können, so sollten Sie zumindest über ein Megabyte freien Speicher verfügen.

Zwei Laufwerke

Am einfachsten läßt sich mit zwei Laufwerken arbeiten, wenn wir eine Kopie der Workbench erstellen, die wir dann an unsere Bedürfnisse anpassen. Dazu legen wir die Workbench in DF0 und eine leere Diskette in DF1 (als Amiga-2000-Besitzer denken Sie bitte daran, daß ein eventuell extern angeschlossenes Laufwerk die Bezeichnung DF2 hat!). Kopieren Sie jetzt die Workbench genau so, als ob Sie eine Sicherheitskopie erstellen wollten. Nun löschen Sie von der soeben erstellten Arbeitskopie der Workbench sämtliche Dateien außer:

```
Trashcan (dir)
c        (dir)
System   (dir)
l        (dir)
devs     (dir)
s        (dir)
t        (dir)
fonts    (dir)
libs     (dir)
.info
CLI
CLI.info
Disk.info
System.info
Trashcan.info
```

Beachten Sie bitte, daß man eine Datei mit "delete <dateiname>" löschen kann, einen ganzen Ordner mit Inhalt aber mit "delete <Ordner> all".

Legen Sie nun die neue Workbench in DF0 und die Lattice-C-Diskette in DF1 (bzw. DF2 beim Amiga 2000).

Führen Sie einen Reset aus, indem Sie CTRL und beide Amiga-Tasten (bzw. CTRL Commodore und Amiga-Taste beim Amiga 500).

Geben Sie bitte nun nacheinander ein:

```
copy DF1:s to DF0:s
cd DF0:s
join startup-sequence install-c as test
delete startup-sequence
delete install-c
rename test to startup-sequence
copy s/make to DF0:
```

Denken Sie bitte daran, falls Sie einen Amiga 2000 mit externem Zweitlaufwerk haben, daß Sie statt DF1 DF2 benutzen.

Wenn Sie jetzt mit Ihrer neuen Arbeitsworkbench Ihren Rechner neu starten, werden alle notwendigen Einstellungen für zwei Laufwerke automatisch vorgenommen.

Ein Laufwerk und 1 MByte

Die wichtigsten Dateien des Systems werden von der Workbench in die RAM-Disk kopiert. Am besten, Sie kopieren gleich den ganzen Ordner mit dem Namen "c". Die Lattice-Diskette legen Sie in Laufwerk "DF0:" (wohin sonst) und legen alle sonstigen Dateien ebenfalls in der RAM-Disk ab.

Da beim Lattice-C Unterverzeichnisse an logische Geräte zugewiesen werden, sollte man diese schon in der Startup-sequence, die beim Booten abgearbeitet wird, entsprechend angeben. So wird dem logischen Gerät "INCLUDE:" der Pfadname zu allen

Include-Dateien zugewiesen. Da auch Disketten beim Amiga einen eigenen Namen besitzen, müssen Sie diesen bei den folgenden Sequenzen einsetzen.

```
assign LC: C-DEV-3.03:c
assign LINK: C-DEV-3.03:c
assign INCLUDE: C-DEV-3.03:include
assign LIB: C-DEV-3.03:lib
```

Hier lautet der Diskettenname "C-DEV-3.03", der sich ja schon beim Kopieren in "copy of C-DEV-3.03" ändern kann. Diesen müssen Sie durch den aktuellen Namen ersetzen. Die Zwischenfiles, die vom Compiler erzeugt werden, können Sie ebenfalls umdirigieren, wenn sie nicht im aktuellen Directory landen sollen. Dazu verwendet man "QUAD:", z.B.

```
assign QUAD: ram:
```

Das Verzeichnis, das durch "INCLUDE:" festgelegt wurde, gilt für alle Include-Anweisungen mittels der Größer-/Kleinerzeichen in

```
#include <dateiname.h>
```

Die Datei muß also über den zugewiesenen Path-Namen an "INCLUDE:" zu erreichen sein. Nachdem diese Installation vorgenommen ist, kann mit dem Compilieren, bzw. mit dem Edieren begonnen werden. Sollten Sie den Editor ED benutzen, so finden Sie alle nötigen Informationen zur Bedienung in Ihrem Amiga-DOS-Handbuch.

Der Compiler

Aufgerufen wird der Compiler über seinen Namen "lc" zusammen mit der angegebenen Sourcedatei. Minimal benötigt man also beispielsweise

```
df0:c/lc hallo
```

Durch die Option "-L" kann dem Compiler gleich noch der Aufruf des Linkers übertragen werden. Der Linker (hier ALINK) kann auch einzeln aufgerufen werden.

Niemand wird sich jedoch die Mühe machen, alle Eingaben jedesmal über die Tastatur einzugeben. Man benutzt zweckmäßigerweise eine MAKE-Datei. Die von mir benutzte Datei namens MAKE ist hier im folgenden abgedruckt:

```
.Key file,opt1,opt2,opt3
;          Compile a C program                    Version 3.00
;          Works with Lattice version 3.02 and above
if not exists <file$t1>.c
   echo "File <file$t1>.c does not exist.  Try again."
   skip END
   endif

LC:lc <opt1> <opt2> <opt3> -iINCLUDE: -iINCLUDE:lattice/ <file$t1>
if not exists "<file$t1>.o"
   echo "Compile failed."
   quit 20
   endif

echo "-- Linking...  <file$t1>.o to <file$t1>"
LINK:alink FROM LIB:Lstartup.obj+<file$t1>.o TO <file$t1> LIB
LIB:lc.lib+LIB:amiga.lib MAP <file$t1>.map
echo "-- done compiling and linking '<file$t1>'. --"
LAB END
date >df0:now
```

Aufgerufen wird sie mit der Zeile:

```
execute make dateiname
```

Eine Endung an die Source ".C" brauchen Sie auch hier nicht anzuhängen. Wenn Sie (wie ich) den Aufwand selbst bei dieser Zeile noch minimieren wollen, so sollten Sie die MAKE-Datei in "M" und das Kommando "execute" in "e" umbenennen. Wie lang dann der Name der Sourcedatei wird, können Sie sich ja später noch überlegen. Da der Compiler und der Linker für das Übersetzen einige Zeit benötigen, sollte man das Multitasking des Amiga nutzen. Setzt man vor die ganze Zeile den Befehl

"run", so wird "im Hintergrund" gearbeitet, und wir könnten dann bereits ein anderes Listing eintippen. Auch dieses Kommando habe ich zur täglichen Arbeit in ein schlichtes "r" umbenannt. Dadurch beschränkte sich die Eingabe auf

```
r e m dateiname
```

Anmerkung: Das Kommando "RUN" muß noch in dieser Schreibweise im "C/"-Verzeichnis existieren, da es vom Lattice aufgerufen wird. Sollten Sie diesen Befehl als Abkürzung verwenden wollen, müssen Sie also zweimal das gleiche Programm unter verschiedenen Dateinamen abspeichern.

Der Linker

Mit dem Programm ALINK steht dem Programmierer ein leistungsfähiger Linker zur Verfügung. Aus der großen Menge an verschiedenen Optionen hier die wichtigsten:

Nach dem Namen "ALINK" gibt man nach dem Parameter "FROM" (wahlweise auch ROOT, oder überhaupt nichts) alle Files an, die zusammenzulinken sind. Anschließend, nach "TO" steht der Name des endgültigen ausführbaren Programmes. Nun können eventuell zu durchforstende Librarydateien nach einem weiteren Parameter "LIBRARY" aufgelistet werden. Einige Beispielaufrufe gefällig:

```
ALINK FROM a,b,c TO programm
ALINK a+b+c TO programm LIBRARY ordner/d
ALINK ROOT a,b,c TO ordner/prg LIBRARY system/lib,obj/special
```

Durch den Parameter "WITH" können Sie alle Optionen nochmals in einer Datei ablegen, genau wie bei einer MAKE-Datei. Ein Linkeraufruf beschränkt sich dann auf

```
ALINK WITH datei
```

In dieser Datei finden sich dann oben aufgeführte Optionen, allerdings jeweils ein Parameter in einer neuen Zeile:

```
ROOT a,b,c
TO ordner/prg
LIBRARY system/lib,obj/special
```

Anhang D: Aztek-C

Der Aztek gibt sich mit einem Laufwerk zufrieden. Ohne größere Einschränkungen lassen sich auch längere Programme recht schnell übersetzen. Bei 1 MByte Speicher kann man sogar alle wichtigen Programme, wie Compiler, Assembler und Linker in die RAM-Disk packen, wodurch sich der ohnehin schon recht schnelle Compiliervorgang noch einmal erheblich beschleunigt.

Den Compiler mit dem Namen "CC" finden Sie im Unterdirectory C/. Der Aufruf ist denkbar einfach:

```
cc datei.c
```

Nun wird der Source "datei.c" compiliert und in Assemblercode übersetzt. Dieser Code kann von einem Maschinensprache-Programmierer noch nach eigenem Ermessen optimiert werden. Diese Datei trägt den Namen "ctmpAXX.XXX", wobei X jeweils eine Ziffer darstellt, die von Aufruf zu Aufruf verschieden ist. Am besten, Sie schauen sich das aktuelle Inhaltsverzeichnis einmal an, denn gleich brauchen Sie diesen Namen.

Auch der Aztek benutzt symbolische Namen für Geräte, die im folgenden aufgelistet sind:

```
CLIB
INCLUDE
CCTEMP
```

Der letzte Name entspricht voll und ganz dem QUAD: beim Lattice. Er bestimmt, wo temporäre Dateien des Compilers abgelegt werden. CLIB gibt den Pfad zu den Bibliotheken an, während in INCLUDE gleiches für die ".H"-Dateien vermerkt wird. Beispiele wären:

```
assign CLIB: df0:lib/
assign INCLUDE: df0:include/
assign CCTEMP: ram:
```

Vor dem Namen der zu compilierenden Datei können diverse Optionen angegeben werden. Hier die wichtigsten:

-Ipath: Durch -I kann ein Path-Name angegeben werden, in dem die Include-Dateien vermutet werden. Es wird nur noch in diesem Unterverzeichnis nach solchen Files gesucht. Die Option ist mit der Zuweisung an INCLUDE (s.o.) vergleichbar.

Bemerkung: Nach dem "i" folgt sofort der Path-Name, ohne zusätzliche Leerzeichen, z.B.:

```
cc -Iram:includes/privat/
```

+C erzeugt einen längeren Code, da Sprünge innerhalb des Programmcodes grundsätzlich mit 32-Bit-Verweisen anstelle von eventuell möglichen 16-Bit-Verweisen realisiert werden.

+D veranlaßt Daten in 32-Bit-Form zu speichern. Dies verlangsamt den Datenzugriff, erhöht den Speicherbedarf, ermöglicht aber (theoretisch) beliebig große Datensegmente. Bei "normalem" Datenzugriff über 16-Bit-Adressierung ist man auf maximal 64 KByte Daten "beschränkt".

+L Variablen und Konstanten vom Typ int werden grundsätzlich in 32 Bit abgespeichert. Dadurch werden die Programme (teilweise) zum Lattice kompatibel, da dort immer 32 Bit verwendet werden. Ohne diese Option reichen für int-Zahlen 16 Bit aus.

-D definiert eine Konstante. Sie entspricht der "#define"-Anweisung, wird aber beim Aufruf des Compilers zugewiesen. Auch hier darf kein Leerzeichen dem Optionsbuchstaben folgen. Beispiel:

```
cc -DTESTLAUF=1 datei.c
```

Die Zuweisung entspricht dem Define

```
#define TESTLAUF 1
```

-S unterdrückt "warnings". Die Warnungen sind nur Hinweise, so daß i.d.R. die compilierten Programme lauffähig sind. Um sich auf wirkliche Fehler zu beschränken, kann man sich durch diese Option lediglich die Fehlermeldungen auf dem Bildschirm ausgeben lassen.

+p veranlaßt den Compiler, einen zum Lattice kompatiblen Code zu erzeugen. Alle Daten, Sprünge und int-Zahlen werden automatisch in der 32-Bit-Version verfaßt.

Der Assembler

Nach dem Compiler kommt der Assembler namens "AS" an die Reihe. Auch er befindet sich im Verzeichnis "C/". Der Aufruf ist ähnlich einfach wie beim Compiler:

```
as datei.o
```

Durch die Option -O kann der entstehenden Datei ein neuer Name mitgegeben werden. Beispiel:

```
as -O programm.o ctmpxyz.123
```

Weitere Optionen wären höchstens für den Assemblerspezialisten interessant. Da wir aber in C programmieren, brauchen wir uns darum keine Gedanken zu machen. Es ist ja lediglich ein Zwischenschritt.

Der Linker

Beim Aztek heißt der mitgelieferte Linker "ln", der sich auch im Subdirectory "C/" befindet. Die zusammenzulinkenden Dateien werden einfach hintereinandergestellt. Ob es sich um Libraries oder Module handelt, ist zwar prinzipiell egal, man sollte aber die Libraries an den Schluß der Liste stellen.

```
ln datei.o c.lib
```

Die Standarddatei "c.lib" befindet sich ebenso wie alle anderen Libraries im Verzeichnis "lib/". Es können so auch mehrere Module zusammengebunden werden:

```
ln -o result modul1.o modul2.o modul3.o c.lib
```

Durch -O erfolgt wieder die Namenszuweisung des resultierenden Programmes. Mit -L können ebenfalls Libraries dem Linker mitgeteilt werden, dann allerdings kann die Endung ".lib" entfallen. Beispiel:

```
ln datei.o -Lc -Lm
```

Zwei weitere interessante Optionen sind +C und +F, wodurch spezielle Speicherbereiche ausgewählt werden können. Hinter der Option folgt ein Kennbuchstabe, der folgende Bedeutung hat:

c	Programm
d	Initialisierte Daten
b	Nicht initialisierte Daten

Das "+C" steht für "chip-memory", "+F" für "fast-memory". Diese zwei Gruppen von RAM-Bereichen sind für die Grafikprogrammierung besonders wichtig, da hier bestimmte Daten immer im Chip-Memory abgelegt werden müssen. Durch diese Option kann man entsprechendes veranlassen:

```
ln +Cdb +Fc datei.o -Lc
```

veranlaßt, die Daten im Chip-Memory und das eigentliche Programm im "normalen" Speicher, dem Fast-Memory, unterzubringen. Ohne besondere Anweisungen werden übrigens alle Informationen im Fast-Memory-Bereich deponiert.

Wie auch beim Lattice, hier eine MAKE-Datei, die auf den Aztek zugeschnitten ist.

```
.key file

echo " Ich kompiliere  <file$t1>.c "
cc -t <file$t1>.c
echo " Ich assembliere  <file$t1>.asm  "
as <file$t1>.asm
echo " Ich linke  <file$t1>.asm zu <file$t1>  "
ln <file$t1>.o -lm -lc ; -lm -lc heißt :linke clib u. mathlib dazu
echo " Alles Klar !"
```

Eine Startup-Sequence, die im RAM alle wichtigen Bestandteile unterbringt, ist im folgenden abgedruckt. Hier wird davon ausgegangen, daß die Aztek-Diskette im Laufwerk "DF2:" untergebracht ist und ein Arbeitsspeicher von mindestens 1 MByte zur Verfügung steht. Diese Startup-Sequence sollten Sie sich dann auf Ihre eigene Anlage maßschneidern.

```
stack 10240
system/setmap d
run newcli
set CLIB=df2:lib/ INCLUDE=df2:include CCTEMP=ram:
echo ""
echo ""
echo "         Welcome to the WONDERFUL world of Aztec C!!"
echo ""
echo "                Version 3.20a   02/27/86"
echo ""
echo "                   by Jim Goodnow II"
echo "     --- Ich installiere die deutsche Tastatur ---"

; Anpassung für include-Files auf 2. Laufwerk

;cc temp = ramdisk (alle Zwischenfiles im RAM)
;INCLUDE = df2:include ( include-Files von df2:)
;clib = df2:lib/       (clib von df2:)

;system/setmap d    Deutsche Tastatur
setdate
echo " <m> ist das Execute-File "
```

```
echo "Ich kopiere C ins RAM"
makedir ram:c
makedir ram:aztek
copy df0:c/copy ram:c
assign c: ram:c

copy df0:c/cc ram:c
copy df0:c/as ram:c
copy df0:c/ln ram:c
copy df2:lib/C.LIB ram:aztek
copy df2:lib/M.LIB ram:aztek
copy df0:c/assign ram:c
copy df0:c/path ram:c
copy df0:c/execute ram:e
copy df0:c/run ram:c
copy ram:c/run ram:c/r
copy df0:c/dir ram:c
copy df0:c/delete ram:c
copy df0:m ram:
copy df0:c/echo ram:c
copy dfo:c/cd ram:c
copy df0:c/ed ram:c

path add df0:c

set CLIB=ram:aztek/ INCLUDE=df2:include CCTEMP=ram:
cd ram:
echo "fertig!"
```

Anhang E: Reservierte C-Befehlswörter

Befehle, die hier aufgeführt, aber nicht im Buch erläutert sind, haben entweder keine Funktion bei derzeitigen C-Compilern oder sind für zukünftige Aufgaben reserviert.

auto	enum	short
break	extern	sizeof
case	float	static
char	for	struct
continue	goto	switch
default	if	typedef
do	int	union
double	long	unsigned
else	register	void
entry	return	while

Anhang F: Prioritäten und Reihenfolge der C-Operatoren

Priorität	Operator	Beschreibung	Auswertung
1	()	Funktion	links nach rechts
	[]	Array	links nach rechts
	.	Strukturverweis	links nach rechts
	->	Strukturverweis (Zeiger)	links nach rechts
2	cast	erzwungene Typumwandlung	rechts nach links
	*	Inhalt von	rechts nach links
	&	Adresse von	rechts nach links
	-	negatives Vorzeichen	rechts nach links
	!	logisches Nicht	rechts nach links
	~	Bitweises Komplement	rechts nach links
	++	Inkrement	rechts nach links
	--	Dekrement	rechts nach links
	sizeof	Speicherbedarf	rechts nach links
3	*	Multiplikation	links nach rechts
	/	Division	links nach rechts
	%	Restdivision (Modulo)	links nach rechts
4	+	Addition	links nach rechts
	-	Subtraktion	links nach rechts
5	>	Shift nach rechts	links nach rechts
	<	Shift nach links	links nach rechts
6	<	kleiner als	links nach rechts
	>	größer als	links nach rechts
	<=	kleiner oder gleich	links nach rechts
	>=	größer oder gleich	links nach rechts
7	==	gleich	inks nach rechts
	!=	ungleich	links nach rechts
8		Bitweises UND	links nach rechts
9	^	Bitweises EXOR	links nach rechts
10	|	Bitweises ODER	links nach rechts
11	&&	Logisches UND	links nach rechts
12	||	logisches ODER	links nach rechts
13	?:	Bedingte Bewertung	rechts nach links
14	=	Zuweisung	rechts nach links
	#=	Verkürzte Zuweisung	rechts nach links
		# aus (+, -, *, /, %, >>, <<, &, |, ~)	
15	,	Trennung von Ausdrücken	links nach rechts

Anhang G: Speicherklassen

Speicherklasse	Gültigkeit	Lebensdauer
auto	Block	Block
extern	Programm	Programm
register	Block	Block
static (intern)	Block	Programm
static (extern)	Datei	Programm

Anhang H: Typumwandlungen

Regeln:

1. char und short werden immer in int und float in double umgewandelt.
2. Sollte nach dieser Umwandlung einer der Operatoren den Typ double haben, so werden der zweite Operand und das Ergebnis ebenfalls in double umgewandelt.
3. Wenn ein Datentyp jetzt long ist, werden alle beteiligten Werte auch in long transformiert.
4. Sollte sich unter den Operanden noch ein unsigned-Wert befinden, erfolgt die Umwandlung aller Werte in unsigned.

Anhang I: Modi für fopen (beim Lattice-C)

String	Neu erzeugen	Datei abschneiden	Lesen	Schreiben	Anhängen	Binär
"r"	nein	nein	ja	nein	nein	ja
"w"	ja	ja	nein	ja	nein	ja
"a"	ja	nein	nein	nein	ja	ja
"r+"	nein	nein	ja	ja	nein	ja
"w+"	ja	nein	ja	ja	nein	ja
"a+"	ja	nein	ja	nein	ja	ja
"ra"	nein	nein	ja	nein	nein	nein
"wa"	ja	ja	nein	ja	nein	nein
"aa"	ja	nein	nein	nein	ja	nein
"ra+"	nein	nein	ja	ja	nein	nein
"wa+"	ja	nein	ja	ja	nein	nein
"aa+"	ja	nein	ja	nein	ja	nein
"rb"	nein	nein	ja	nein	nein	ja
"wb"	ja	ja	nein	ja	nein	ja
"ab"	ja	nein	nein	nein	ja	ja
"rb+"	nein	nein	ja	ja	nein	ja
"wb+"	ja	nein	ja	ja	nein	ja
"ab+"	ja	nein	ja	nein	ja	ja

Bei binären Dateien werden keine Umwandlungen vorgenommen. Wird das File als ASCII-Datei geöffnet, das ist durch das "a" an der zweiten Stelle zu erkennen, so werden beim Lesen alle Carriage-Returns (Code 13 = '\r') eliminiert und das Zeichen mit dem Code (26) in EOF (-1) umgewandelt. Beim Schreiben wiederum entsteht aus einem einzelnen Line-Feed ('\n') die Zeichenkombination "\r\n".

Lattice-C verwendet zur Unterscheidung der beiden Modi eine externe int-Variable mit Namen "_fmode". Sollte das höchstwertige Bit gesetzt sein (_fmode & 0x8000), so wird der binäre Modus verwendet, ansonsten werden die angegebenen Umwandlungen durchgeführt.

Änderungen beim Aztek:

Der Aztek öffnet alle Dateien binär. Zusätzlich bietet er aber den Modus "x" und "x+" an, der eine Datei zum Schreiben öffnet. Sollte dieses File noch nicht existieren, so wird es kreiert. Durch "x+" kann man nach dem Öffnen der Datei diese nicht nur beschreiben sondern auch lesen.

Anhang J: Stichwortverzeichnis

& 125, 157
* 127
-> 163
. 163
<< 159
>> 159
^ 160
| 157
~ 160

Abbruchkriterium 246
Absturz 134
ACCESS_READ 255
Activate 228
Adreß-Operator 125
Adresse 125
Amiga-DOS 253
Anweisungsblock 124
Append 194
Argc 188
Auflösung 235
Aztek 233

Backslash 77
Bibliotheken 20, 225
Binärsystem 155
Bindings 144
Bitmap 235
Bitplanes 235, 247
Bits 155, 166
Bitverknüpfung 155
BlockPen 236

C-Compilers 225
Case 124
Cast-Anweisung 233
Char 72
Close-Gadget 239

CloseWindow 231, 239
Compiler 14, 15, 19
COSTOMSCREEN-Define 236
Creat 203

Dateiende 206
Dateizeiger 205, 206
Datenmengen 198
Datentypen 74, 131, 163, 167
Default 124
DefaultTitle 236
Defines 151
Definition 31
Deklaration 107, 170
Delay-Funktion 253
Depth 236
DetailPen 236
Dezimalzahlen 91
Dimensionen 148
Directory 255
Division 66, 159
Doppelpunkt 161
DOS 253
Dos.library 253
Draw 241
Dualsystem 158

ED 23
Element 132
Endekennzeichen 130
Examine 255
Extern 137

Fakultät 52
Farben 227, 234
Farbregister 247, 227
Fclose 194
Fehler 161
Fehlerdatei 19

Fehlermeldungen 19
Feldbreite 79, 166
Felder 114
File-Handle 203
FILE-Pointer 192
FileInfoBlock 255
FILENOTE 254
Float 70
Font 245
Fopen 193
Formatanweisungen 36, 77
Formatierung 29
Fread 198
Fseek 205
Ftell 207
Funktionen 20, 225
Funktionsdefinition 105
Fwrite 198

Genauigkeit 70
Geschwindigkeitsvorteil 138
Getc 192
Gleichheitszeichen 52, 66
Gleitkommazahlen 70
Graphics.library 241

Hauptdirectory 143
Hintergrundfarbe 228
HIRES 249

IDCMPFlags 239
Include-Files 234
Index 113, 115, 134
Inhaltsverzeichnis 255
Initialisierung 110, 120, 133, 147, 165
Integerwerte 35
Interlace 247
Interpreter 15
Intuition 225

Intuition-Library 226, 231
Intuition/intuition.h 227

Kanal 192
Kanalnummer 202
Klammern 103, 222
Kommentare 33
Kopierprogramm 195
Kurzform 97

Label 161
Lebensdauer 135
Lesen 194
Library 225
Linie 241
Lock 255
Logisch wahr 120
Long 72
Lseek 205

Main 28
MAKE-Datei 21
Makro 180
Makros mit Parameterübergabe 180
Maske 157
Maus-Koordinaten 246
MaxHeight 230
Maximalwerte 223
Maximum 153, 180
MaxWidth 230
MinHeight 230
Minimum 153
Minuszeichen 79
MinWidth 230
Modul 20
Modulo 66, 149
MouseX 241
MouseY 241
Move 240
Multiplikation 66

Nachkommastellen 79
NewScreen 235
NewWindow 227
Nocarerefresh 229

Objectcode 20
Öffnen eines Windows 230
Open 202, 203
OpenWindow 230

PAL-Amiga 230
Parameterdeklarationen 106
Parser 8
Pixelbearbeitung 249
Pointer 127
Präprozessor 8, 182
Printf 28, 32, 78, 183
Prioritäten 164, 172, 181
Programmabsturz 224
Programmlänge 180
PROTECT 257
Prozentzeichen 77
Prozessor 137
Puffer 191, 198
Putc 192

RastPort-Struktur 240
Rechnerabsturz 71
Referenz 127
Rename 254
Reverse 151
Rückgabewert 253
Rundung 67

Scanf 78
Scanner 8
Schleife 51
Screen-Programm 237
Screen-Struktur 236

Screens 234
Seiteneffekte 182
Semikolon 26, 106, 222
SetComment 254
Short 72
Sizeof 179, 199
Smart Refresh 228
Speicherplatz 135, 166, 178
Sprungbefehl 162
Stdio.h 192
Stern 127
Steuerzeichen 32
Strcmp 145
Strcpy 108, 130
Stringende 134
Struct 164
Struct NewWindow 226
Strukturen 226
Subdirectories 143
Systemabsturz 233
System-Gadgets 229

Tabulatorstops 76
Tausch-Funktion 128
Tell 207
Text 240
Title 229
TOPAZ_SIXTY 246
Typumwandlungen 74

Übergabeparameter 105, 107, 128
Unabhängig 180
UNIX 7
Unsigned 72
Unterstreichungszeichen 68

Variablen 35
Versionsnummer 233
Verweis 127
Void 107

Warnings 233
WBENCHSCREEN 236
Wertigkeit 157
Wertzuweisungen 81
Window 226
Window-Flags 228
Window-Programm 231
WINDOWCLOSE 239
Windowdepth 229
Windowrag 229
Windowsizing 228
Wort 155
WritePixel 250

Zeiger 127, 226
Zusätze 78

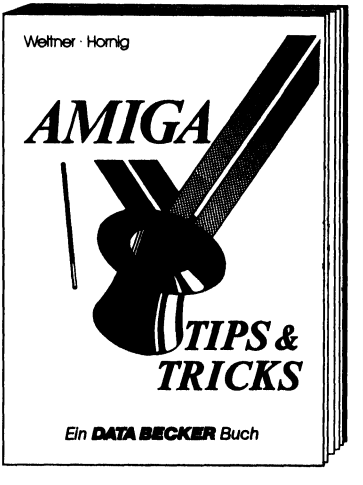
AmigaBASIC für alle. Im ersten Teil werden Sie Schritt für Schritt – und vor allem auf verständliche Art und Weise – in die Programmierung des AMIGA eingeführt: Grafik und Sound gehören genauso dazu wie Datenverwaltung und Statistik. Im zweiten Teil finden Sie alle gelernten Befehle mit Syntax- und Parameterangaben zum schnellen Nachschlagen. Dazu gibt es Programme und Utilities in Hülle und Fülle.

Aus dem Inhalt:

- Das Videotitel-Programm zeigt die OBJECT-Animation
- Das Balken- und Tortengrafik-Programm erklärt die Grafikbefehle
- Das Malprogramm mit Windows, Pulldowns, Mausbefehlen, Füllmustern, Einlesen und Abspeichern von IFF-Bildern
- Das Statistikdaten-Programm hilft, sequentielle Dateien zu verstehen
- Die Datenbank zeigt den Umgang mit relativen Dateien
- Das Sprachutility sorgt für mehr Verständnis bei der Sprachprogrammierung
- Das Synthesizer-Programm führt Sie in die Welt der Töne, Wellenformen und Hüllkurven

Rügheimer, Spanik
AmigaBASIC
Hardcover, 775 Seiten, DM 59,–
ISBN 3-89011-209-9

Amiga Tips und Tricks ist eine riesige Fundgrube für den Amiga-Besitzer. Viele Beispielprogramme in BASIC und C zeigen, wie man die fantastischen Möglichkeiten dieses Superrechners optimal nutzen kann. Und ganz nebenbei lernt man noch eine Menge über den Aufbau des Computers und seine Programmierung.

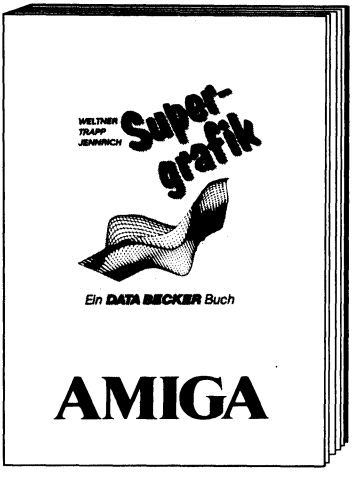


Aus dem Inhalt:

- Nutzung der wichtigsten Libraries von BASIC aus: Graphic, DOS, Exec, Intuition
- Nutzung der verschiedenen Disk-Fonts in BASIC-Programmen
- Verschiedene Schrifttypen in BASIC-Programmen: Bold, Outline, Shadow
- Zugriff auf das CLI von BASIC aus
- Bewegbare Screens und Windows mit eigenen Titeln
- Intuition in eigenen Programmen nutzen: Autorequest, Guru Meditation
- Gesamte Directory-Struktur ausdrucken
- Ein-/Ausgabehandling: Diskmonitor, Hardcopy von Windows und Screen
- Speicherverwaltung: AllocMem und FreeMem
- Filehandling in C: Anzahl freier Blöcke, File exist Prüfung, Filegröße, File-kommentar, Get Protection Prüfung
- Zugriff auf Intuition am Beispiel eines einfachen Grafikprogramms: Screen, Windows, Menue
- Half Bright und Interlace Modus
- Druckerhandling in C

Weltner/Hornig
AMIGA Tips & Tricks
Hardcover, ca. 364 Seiten, DM 49,–
ISBN 3-89011-211-0

Der Amiga ist eine tolle Grafik-Maschine. 4096 Farben gleichzeitig, 640 × 400 Punkte Auflösung, Sprites, Bobs und die Geschwindigkeit des Blitters begeistern einfach. Das AMIGA SUPERGRAFIK-Buch zeigt Ihnen, wie Sie diese tollen Features in den Griff bekommen. Dabei ist es gleichgültig, ob Sie mit BASIC einsteigen, über die Amiga-Libraries die schnellen Routinen von BASIC aus nutzen oder komplette Programme in C schreiben wollen: Dieses Buch zeigt mit vielen Beispielprogrammen, wie man die Grafik des Amiga programmiert.



Aus dem Inhalt:

- Die Grafik-Befehle des AmigaBASIC (Punkt, Linie, Kreis, Rechteck, Muster, Flächen füllen)
- Laden und Speichern von Grafiken (IFF-Format)
- Sprites, Bobs, Animation
- CAD durch 1024 × 1024-Superbitmap
- Verschiedene Zeichensätze und Schriftarten in BASIC
- Neue Möglichkeiten von BASIC aus durch Zugriff auf die Libraries und Spezial-Chips (4096 Farben gleichzeitig, farbige Muster, Hardcopy von Screens und Windows)
- Grafikprogrammierung von C aus (Punkt, Linie, Rechtecke, Polygone, Farben)
- Die Routinen der Grafik-Bibliothek nutzen
- Das Animationssystem des Amiga (Sprites, Bobs, AnimObs)
- Programmierung von Copper und Blitter (Rasterzeilen-Interrupt, blitzschnelles Kopieren)
- Komplette Beschreibung des Amiga-Grafiksystems (View, Viewport, RastPort, Aufbau der Bitmaps, Screens, Windows)

Weltner/Trapp/Jenrich
Supergrafik
686 Seiten, DM 59,–
ISBN 3-89011-254-4

DAS STEHT DRIN:

Wer auf dem Superrechner AMIGA schnell in die Supersprache C einsteigen will, der findet in diesem Buch den schnellen Einstieg: Mit „C an einem Wochenende". Dieser Einführungskurs löst die Anfangsschwierigkeiten und führt gleichzeitig in die Bedienung der beiden wichtigsten vorhandenen Compiler ein. Anschließend wird der gesamte Sprachumfang mit vielen Beispielen erläutert und der Umgang mit den Routinen der C-Bibliotheken erklärt. Den krönenden Abschluß bildet dann eine Einführung in die Programmierung des Amiga-Betriebssystems.

Aus dem Inhalt:

- Das Besondere an C
- Bedienung eines C-Compilers
- Das erste Programm
- Der Sprachumfang von C (Schleifen, Bedingungen, Funktionen, Strukturen)
- Die speziellen Fähigkeiten von C
- Die wichtigsten Routinen der C-Bibliotheken
- Ein-/Ausgabe
- Tips und Tricks zur Fehlersuche
- Einstieg in die direkte Programmierung des Betriebssystems (Windows, Screens, direkte Textausgabe, DOS-Funktionen)
- Arbeit mit Lattice C und dem Aztek-Compiler

UND GESCHRIEBEN HAT DIESES BUCH:

Dirk Schaun ist C- und Grafikspezialist bei DATA BECKER. Er beschäftigt sich seit 4 Jahren mit dem Computer. Seit es den AMIGA gibt, rückt er ihm mit C zu Leibe. Trotz seiner Professionalität hat er die Probleme der Einsteiger nie aus den Augen verloren.

ISB N 3-89011-107-6 DM +039.00

DM 39,–
ÖS 304,–
sFr 37,–

DATA BECKER

9 783890 111070

03900